Selenium 中文文档

wizardforcel

Published with GitBook

目錄

selenium 中文文档

译者：fool2fish

来源：selenium-doc

这里主要集中了 Selenium 官网的所有文档，和项目 Wiki 中的部分文档翻译。

希望这些文档能帮助大家更好的了解 Selenium 的工作原理，而不仅仅是会使用那些客户端驱动的 API.

本人翻译完成大部分官网文档后处于停歇状态，期待你的提交：）

译文

官网文档

- 介绍
- Selenium WebDriver
- Selenium 1 (Selenium RC)
- Selenium Grid

wiki

- WebDriverJS
- 高级用户交互

如何参与

- 如有发现 bug 或者有任何建议，请提 issue 。
- 你也可以直接 fork 本项目，修改或添加内容，然后 pull request 。

介绍

用于网站应用的测试自动化

如今，大多数软件应用都是跑在浏览器中的网站应用。不同公司和组织之间的测试效率迥异。在这个富交互和响应式处理随处可见的时代，很多组织都使用敏捷的方式来开发，因此测试自动化也成为软件项目的必备部分。测试自动化意味着使用软件工具来反复运行项目中的测试，并为回归测试提供反馈。

测试自动化有很多优点。大多数都和测试的可重复性和高执行效率这两点相关。市面上有一些商业或开源的同居来辅助测试自动化开发。Selenium 应该是最广泛使用的开源方案。本文档将帮助新手和有经验的使用者学习为网站应用创建测试自动化的有效技术。

本文档介绍了 Selenium，其细节和从社区中积累的最佳实践。其中包含很多范例。同时，也将提及 Selenium 的一些技术细节和推荐用法。

对于一个软件团队的测试过程来说，测试自动化具有提高长期效率的优势。测试自动化包括：

- 频繁的回归测试
- 快速反馈
- 几乎无限制的测试用例迭代执行
- 支持敏捷和极限编程
- 遵循测试用例的文档
- 自定义缺陷报告
- 能找出手工测试中没发现的缺陷

自动化？或不自动化？

自动化是否总是好的？什么时候我们应该使用自动化的方式来测试？

自动化测试不总是有优势的。这里有一些场景就更适合手工测试。例如，如果一个应用的接口在不久的将来会发生变化，那时所有的测试用例都需要重写。有时仅仅是因为没有足够的时间来实现测试自动化。短期来说，手工测试更快捷。如果一个应用的发布日是掐死的，而又没有可用的自动化测试，而测试工作又必须在指定时间内完成，那么此时手工测试也是最佳选择。

介绍 **Selenium**

Selenium 是一组软件工具集,每一个都有不同的方法来支持测试自动化。大多数使用 Selenium 的QA工程师只关注一两个最能满足他们的项目需求的工具上。然而，学习所有的工具你将有更多选择来解决不同类型的测试自动化问题。这一整套工具具备丰富的测试功能，很好的契合了测试各种类型的网站应用的需要。这些操作非常灵活，有多种选择来定位 UI 元素，同时将预期的测试结果和实际的行为进行比较。Selenium 一个最关键的特性是支持在多浏览器平台上进行测试。

Selenium 项目简史

Selenium 诞生于 2004 年，当在 ThoughtWorks 工作的 Jason Huggins 在测试一个内部应用时。作为一个聪明的家伙，他意识到相对于每次改动都需要手工进行测试，他的时间应该用得更有价值。他开发了一个可以驱动页面进行交互的 Javascript 库，能让多浏览器自动返回测试结果。那个库最终变成了 Selenium 的核心，它是 Selenium RC（远程控制）和 Selenium IDE 所有功能的基础。Selenium RC 是开拓性的，因为没有其他产品能让你使用自己喜欢的语言来控制浏览器。

Selenium 是一个庞大的工具，所以它也有自己的缺点。由于它使用了基于 Javascript 的自动化引擎，而浏览器对 Javascript 又有很多安全限制，有些事情就难以实现。更糟糕的是，网站应用正变得越来越强大，它们使用了新浏览器提供的各种特性，都使得这些限制让人痛苦不堪。

在 2006 年，一名 Google 的工程师， Simon Stewart 开始基于这个项目进行开发，这个项目被命名为 WebDriver。此时，Google 早已是 Selenium 的重度用户，但是测试工程师们不得不绕过它的限制进行工具。Simon 需要一款能通过浏览器和操作系统的本地方法直接和浏览器进行通话的测试工具，来解决Javascript 环境沙箱的问题。WebDriver 项目的目标就是要解决 Selenium 的痛点。

跳到 2008 年。北京奥运会的召开显示了中国在全球的实力，大规模的次贷危机引发了“大萧条”以来美国最大的经济危机。但是当年最重要的故事是 Selenium 和WebDriver 的合并。Selenium 有着丰富的社区和商业支持，但 WebDriver 显然代表着未来的趋势。两者的合并为所有用户提供了一组通用功能，并且借鉴了一些测试自动化领域最闪光的思想。或许，关于两者合并的最好解释，是由 WebDriver 的开发者，在 2009 年 8 月 6 日发出的一封给社区的联合邮件中提到的：

> 为什么这两个项目要合并？一部分是因为 WebDriver 弥补了 Selenium 的一些短处（例如提供了一组很棒的 API，绕开浏览器的限制），一部分是因为 Selenium 弥补了 WebDriver 的一些短处（例如对浏览器更广泛的支持），还有一部分是因为 Selenium 的主要贡献者和我都认为这样能为用户提供最优秀的框架。

Selenium 工具集

Selenium 由多个软件工具组成，每个具备特定的功能。

Selenium 2 (又叫 Selenium Webdriver)

Selenium 2 代表了这个项目未来的方向，也是最新被添加到 Selenium 工具集中的。这个全新的自动化工具提供了很多了不起的特性，包括更内聚和面向对象的 API，并且解决了旧版本限制。

正如简史中提到的，Selenium 和 WebDriver 的作者都赞同两者各具优势，而两者的合并使得这个自动化工具更加强健。

Selenium 2.0正是于此的产品。它支持WebDriver API及其底层技术，同时也在WebDriver API底下通过Selenium 1技术为移植测试代码提供极大的灵活性。此外，为了向后兼容，Selenium 2 仍然使用 Selenium 1 的 Selenium RC 接口。

Selenium 1 (又叫 Selenium RC 或 Remote Control)

正如你在简史中读到的，在很长一段时间内，Selenium RC 都是最主要的 Selenium 项目，直到 WebDriver 和 Selenium 合并而产生了最新且最强大的 Selenium 2.

Seleinum 1 仍然被活跃的支持着（更多是维护），并且提供一些 Selenium 2 短时间内可能不会支持的特性，包括对多种语言的支持(Java, Javascript, Ruby, PHP, Python, Perl and C#) 和对大多数浏览器的支持。

Selenium IDE

Selenium IDE (集成开发环境) 是一个创建测试脚本的原型工具。它是一个 Firefox 插件，提供创建自动化测试的建议接口。Selenium IDE 有一个记录功能，能记录用户的操作，并且能选择多种语言把它们导出到一个可重用的脚本中用于后续执行。

注意

虽然 Selenium IDE 有保存功能，能让用户以表格的形式保存测试，以供后续的导入和执行，但它不是用于执行你的测试是否通过，也不能创建所有你需要的自动化测试。需要注意的是，Selenium IDE 不能生成含有迭代和条件语句的测试脚本。在本文档编写时也没有要实现该功能的计划。这部分是因为技术原因，部分是因为 Selenium 的开发者所推荐的自动化测试的最佳实践常常是需要编写一些代码的。Selenium IDE 只是被设计为一个快速的原型工具。Selenium 的开发者推荐选用支持的最好的语言来创建严谨、健壮的测试，不管是使用 Selenium 1 还是 Selenium 2.

Selenium-Grid

Selenium-Grid 使得 Selenium RC 解决方案能提升针对大型的测试套件或者哪些需要运行在多环境的测试套件的处理能力。Selenium Grid 能让你并行的运行你的测试，也就是说，不同的测试可以同时跑在不同的远程机器上。这样做有两个有事，首先，如果你有一个大型的测试套件，或者一个跑的很慢的测试套件，你可以使用 Selenium Grid 将你的测试套件划分成几份同时在几个不同的机器上运行，这样能显著的提升它的性能。同时，如果你必须在多环境中运行你的测试套件，你可以获得多个远程机器的支持，它们将同时运行你的测试套件。在每种情况下，Selenium Grid 都能通过并行处理显著地缩短你的测试套件的处理时间。

选择合适你的 Selenium 工具

很多人都从 Selenium IDE 开始学习使用，如果你不是特别善于编程或者编写一门脚本语言，你可以通过使用 Selenium IDE 来熟悉 Selenium 命令。使用 IDE，你能在很短的时间内（有时是数秒）创建简单的测试。

但是我们不推荐你使用 Selenium IDE 来处理所有的测试自动化工作。更高效的做法是，你需要使用它支持的语言创建和运行你的测试，无论是 Selenium 1 还是 Selenium 2。至于选择什么语言则取决于你的喜好。

在编写本文档时，Selenium 的开发者认为Selenium-WebDriver API 才是 Selenium未来的趋势。但Selenium 1 提供向后兼容。同时，我们也在之前讨论了两者的优势和劣势。

我们强烈建议那些初次接触 Selenium 的用户通读这这个章节的内容。那些第一次使用 Selenium ，随意创建了一些测试套件的用户，你通常会希望从 Selenium 2 开始，因为这部分是 Selenium 在将来都会持续支持的。

支持的浏览器和平台

在 Selenium 2.0 中，支持的浏览器完全取决于你是否使用 Selenium-WebDriver 或 Selenium-RC。

Selenium-WebDriver

Selenium-WebDriver 支持如下浏览器，在所有支持这些浏览器的操作系统中能都运行良好。

- Google Chrome 12.0.712.0+
- Internet Explorer 6, 7, 8, 9 - 32 and 64-bit where applicable
- Firefox 3.0, 3.5, 3.6, 4.0, 5.0, 6, 7
- Opera 11.5+
- HtmlUnit 2.9
- Android – 2.3+ for phones and tablets (devices & emulators)
- iOS 3+ for phones (devices & emulators) and 3.2+ for tablets (devices & emulators)

注意：

在写本文档的时候，一款 Android 2.3 的模拟器被报有bug。但是在 tablet 模拟器和真实设备中均工作良好。

Selenium 1.0 and Selenium-RC

这里是指老的，支持 Selenium 1 的部分。它也适用于 Selenium 2 版本中的 Selenium RC。

Browser	Selenium IDE	Selenium 1 (RC)	Operating Systems
Firefox 3.x	Record and playback tests	Start browser, run tests	Windows, Linux, Mac
Firefox 3	Record and playback tests	Start browser, run tests	Windows, Linux, Mac
Firefox 2	Record and playback tests	Start browser, run tests	Windows, Linux, Mac
IE 8	Test execution only via Selenium RC*	Start browser, run tests	Windows
IE 7	Test execution only via Selenium RC*	Start browser, run tests	Windows
IE 6	Test execution only via Selenium RC*	Start browser, run tests	Windows
Safari 4	Test execution only via Selenium RC	Start browser, run tests	Windows, Mac
Safari 3	Test execution only via Selenium RC	Start browser, run tests	Windows, Mac
Safari 2	Test execution only via Selenium RC	Start browser, run tests	Windows, Mac
Opera 10	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Opera 9	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Opera 8	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Google Chrome	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Others	Test execution only via Selenium RC	Partial support possible**	As applicable

* 在 Firefox 上通过 Selenium IDE 开发的测试，可以通过简单的 Selenium RC 命令行在任意支持的浏览器上运行。

** Selenium RC 服务器能开启任何可运行的测试。但根据浏览器的安全设置，可能会有部分特性不可用。

灵活性和可扩展性

你将发现 Selenium 是高度灵活的。你有很多方式为 Selenium 的测试脚本和 Selenium 框架添加功能来定制你的自动化测试。同其他的自动化工具相比，Selenium 可能是最强的。定制相关的内容贯穿整个文档有多处提及。另外，Selenium 是开源的，它的源码可以下载和修改。

本文档包含哪些内容？

本文档同时面向于新手和那些希望了解更多的 Selenium 用户。我们向新手介绍 Selenium，我们并不要求你对 Selenium 非常了解，但你至少需要知道一些自动化测试的基本知识。对于那些经验丰富的用户来说，本文档可作为一个使用参考。如果真的非常熟悉，我们建议你浏览一下每个章节和其副标题。我们提供了 Selenium 的架构信息，常见用法的例子和一章关于测试设计的内容。

剩下的章节将讲述以下内容:

Selenium IDE

介绍 Selenium IDE 以及如何使用它创建测试脚本。如果你缺乏编程经验，但仍然希望学习测试自动化，那么从本章开始入手是个不错的主意，并且你将会发现自己能通过 Selenium IDE 创建不少的测试用例。如果你编程经验丰富，但你希望使用 Seleinum IDE 快速创建测试原型的话，这一章对你也很有用。本章还将向你演示如何导出指定语言的测试脚本，以添加更多 Selenium IDE 不能支持的功能。

Selenium 2

解释如何通过 Selenium 2 创建自动化测试项目。

Selenium 1

解释如何通过 Selenium RC API 开发一个自动化测试项目。我们使用了多种语言来进行代码演示。同时包括了如何安装 Selenium RC 的内容。Seleium RC 支持的各种模式、配置也将会介绍，包括它们的限制和如何进行权衡。我们还提供了架构图来帮助演示这些点。对于

Selenium RC 新手来说，一些常见问题的解决方案也列在其中，例如，操作安全证书，HTTPS 请求，弹出框和打开新窗口。

测试设计

这个章节介绍了使用 Selenium WebDriver 和 Selenium RC 的编程技巧。我们还演示了论坛中常被问道的技巧，例如如何设计 setup 和 teardown 方法，如何实施数据驱动测试（每次测试通过数据都有数据发生变化）和其他一些常见的测试自动化任务的编程方法。

Selenium-Grid

这部分内容未完成。

User extensions

讲述如何修改、扩展和定制 Selenium。

Selenium WebDriver

注意：本章内容官方团队正在完善中。

介绍 WebDriver

Selenium 2.0 最主要的一个新特性就是集成了 WebDriver API。WebDriver 提供更精简的编程几口，以解决 Selenium-RC API 中的一些限制。WebDriver 为那些页面元素可以不通过页面重新加载来更新的动态网页提供了更好的支持。WebDriver 的目标是提供一套精心设计的面向对象的 API 来更好的支持现代高级 web 应用用的测试工作。

同 Selenium-RC 相比，WebDriver 如何驱动浏览器的？

Selenium-WebDriver 直接通过浏览器自动化的本地接口来调用浏览器。如何直接调用，和调用的细节取决于你使用什么浏览器。本章后续的内容介绍了每个 “browser driver” 的详细信息。

相比 Selenium-RC ，WebDriver 确实非常不一样。Selenium-RC 在所有支持的浏览器中工作原理是一样的。它将 JavaScript 在浏览器加载的时候注入浏览器，然后使用这些 JavaScript 驱动 AUT 运行 WebDriver 使用的是不同的技术，再一次强调，它是直接调用浏览器自动化的本地接口。

WebDriver 和 Selenium-Server

你可能需要，也可能不需要 Selenium Server，取决于你打算如何使用 Selenium-WebDriver。如果你仅仅需要使用 WebDriver API，那就不需要 Selenium-Server。如果你所有的测试和浏览器都在一台机器上，那么你仅需要 WebDriver API。WebDriver 将直接操作浏览器。

在有些情况下，你需要使用 Selenium-Server 来配合 Selenium-WebDriver 工作，例如：

- 你使用 Selenium-Grid 来分发你的测试给多个机器或者虚拟机。
- 你希望连接一台远程的机器来测试一个特定的浏览器。
- 你没有使用 Java 绑定（例如 Python, C#, 或 Ruby），并且可能希望使用 HtmlUnit Driver。

设置一个 Selenium-WebDriver 项目

安装 Selenium 意味着当你创建一个项目，你可以在项目中使用 Selenium 开发。具体怎么做取决于你的项目语言和开发环境。

Java

创建一个 Selenium 2.0 Java 项目最简单的方式是使用 maven。Maven 将下载 Java 绑定（Selenium 2.0 的 Java 客户端）和其所有依赖，并且通过 pom.xml（mvn项目配置）为你创建项目。当你完成这些操作的时候，你可以将 maven 项目导入到你偏好的 IDE 中，例如 IntelliJ IDEA 或 Eclipse。

首先，创建一个用于放置项目的文件夹。然后，在这个文件夹中创建 pom.xml 文件，内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.or
    <modelVersion>4.0.0</modelVersion>
    <groupId>MySel20Proj</groupId>
    <artifactId>MySel20Proj</artifactId>
    <version>1.0</version>
    <dependencies>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>2.28.0</version>
        </dependency>
        <dependency>
            <groupId>com.opera</groupId>
            <artifactId>operadriver</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>com.opera</groupId>
                <artifactId>operadriver</artifactId>
                <version>1.1</version>
                <exclusions>
                    <exclusion>
                        <groupId>org.seleniumhq.selenium</groupId>
                        <artifactId>selenium-remote-driver</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

确保你指定了最新版本。在编写本文档时，范例代码中的即为最新版本。但是，稍后 Selenium 2.0 还会不断有新发布。检查 Maven 下载页面 中的最新版本，并修改上述文件中依赖的版本。

命令行进入本目录，运行如下命令：

```
mvn clean install
```

该命令会下载 Selenium 和其所有依赖，并添加到这个项目中。

最后，将项目导入到你的 IDE。对于不太熟悉 IDE 的用户，我们提供了附件来说明相关内容。

Importing a maven project into IntelliJ IDEA

Importing a maven project into Eclipse

从 Selenium 1.0 迁移

对于那些已经使用 Selenium 1.0 编写测试套件的用户，我们提供了一些迁移的建议。Selenium 2.0 的核心工程师 Simon Stewart 写了一篇关于从 Selenium 1.0 迁移的文章，包含在本文的附件中。

Migrating From Selenium RC to Selenium WebDriver

实例介绍 Selenium-WebDriver API

WebDriver 是一个进行 web 应用测试自动化的工具，主要用于验证它们的行为是否符合期望。WebDriver 的目标是提供一套易于掌握的 API，且比 Selenium-RC (1.0) 更易于使用，页能是你的测试更具可读性和维护性。它没有同任何特定的测试框架进行绑定，所以可以在单元测试或者是 main 方法中工作良好。本小节介绍 WebDriver API，并且帮助你熟悉它。如果你还没有任何 WebDriver 项目，请按照上一小节的介绍新建一个。

建好项目后，你可以发现 WebDriver 和任何普通的库一样：它是自包含的，通常不需要进行任何额外的处理或者运行安装。这一点和 Selenium-RC 的代理服务器是不一样的。

注意：使用 Chrome Driver、 Opera Driver、Android Driver 和 iPhone Driver 是需要一些额外操作的。

我们准备了一个简单的例子：在 Google 上搜索 “Cheese”，然偶输出搜索结果页的页面标题到 console。

```
package org.openqa.selenium.example;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.WebDriverWait;

public class Selenium2Example  {
    public static void main(String[] args) {
        // 创建了一个 Firefox driver 的实例
        // 注意，其余的代码依赖于接口而非实例
        WebDriver driver = new FirefoxDriver();

        // 使用它访问 Google
        driver.get("http://www.google.com");
        // 同样的事情也可以通过以下代码完成
        // driver.navigate().to("http://www.google.com");

        // 找到搜索输入框
        WebElement element = driver.findElement(By.name("q"));

        // 输入要查找的词
        element.sendKeys("Cheese!");

        // 提交表单
        element.submit();

        // 检查页面标题
        System.out.println("Page title is: " + driver.getTitle());

        // Google 搜索结果由 JavaScript 动态渲染
        // 等待页面加载完毕，超时时间设为10秒
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.getTitle().toLowerCase().startsWith("cheese!");
            }
        });

        //应该能看到: "cheese! - Google Search"
        System.out.println("Page title is: " + driver.getTitle());

        //关闭浏览器
        driver.quit();
    }
}
```

在接下来的章节中，你将学习到更多使用 WebDriver 的知识，例如根据浏览器历史记录前进和后退，如何测试 frames 和 windows。针对这些点我们提供了全面的讨论和范例。

Selenium-WebDriver API 和操作

获取一个页面

访问一个页面或许是使用 WebDriver 时你第一件想要做的事情。最常见的是调用 “get” 方法：

```
driver.get("http://www.google.com");
```

包括操作系统和浏览器在内的多种因素影响，WebDriver 可能会也可能不会等待页面加载。在某些情况下，WebDriver可能在页面加载完毕前就返回控制了，甚至是开始加载之前。为了确保健壮性，你需要使用 Explicit and Implicit Waits 等到页面元素可用。

查找 UI 元素（web 元素）

WebDriver 实例可以查找 UI 元素。每种语言实现都暴露了 “查找单个元素” 和 “查找所有元素” 的方法。第一个方法如果找到则返回该元素，如果没找到则抛出异常。第二种如果找到则返回一个包含所有元素的列表，如果没找到则返回一个空数组。

“查找” 方法使用了一个定位器或者一个叫 “By” 的查询对象。“By” 支持的元素查找策略如下：

By id

这是最高效也是首选的方法用于查找一个元素。UI 开发人员常犯的错误是，要么没有指定 id，要么自动生成随机 id，这两种情况都应避免。及时是使用 class 也比使用自动生成随机 id 要好的多。

HTML:

```
<div id="coolestWidgetEvah">...</div>
```

Java：

```
WebElement element = driver.findElement(By.id("coolestWidgetEvah"));
```

By Class Name

"class" 是 DOM 元素上的一个属性。在实践中，通常是多个 DOM 元素有同样的 class 名，所以通常用它来查找多个元素。

HTML:

```
<div class="cheese"><span>Cheddar</span></div><div class="cheese"><span>Gouda</span></div
```

Java：

```
List<WebElement> cheeses = driver.findElements(By.className("cheese"));
```

By Tag Name

根据元素标签名查找。

HTML:

```
<iframe src="..."></iframe>
```

Java：

```
WebElement frame = driver.findElement(By.tagName("iframe"));
```

By Name

查找 name 属性匹配的表单元素。

HTML:

```
<input name="cheese" type="text"/>
```

Java：

```
WebElement cheese = driver.findElement(By.name("cheese"));
```

By Link Text

查找链接文字匹配的链接元素。

HTML：

```
<a href="http://www.google.com/search?q=cheese">cheese</a>>
```

Java：

```
WebElement cheese = driver.findElement(By.linkText("cheese"));
```

By Partial Link Text

查找链接文字部分匹配的链接元素。

HTML:

```
<a href="http://www.google.com/search?q=cheese">search for cheese</a>>
```

Java：

```
WebElement cheese = driver.findElement(By.partialLinkText("cheese"));
```

By CSS

正如名字所表明的，它通过 css 来定位元素。默认使用浏览器本地支持的选择器，可参考 w3c 的 css 选择器。如果浏览器默认不支持 css 查询，则使用 Sizzle。ie6、7 和 ff3.0 都使用了 Sizzle。

注意使用 css 选择器不能保证在所有浏览器里都表现一样，有些在某些浏览器里工作良好，在另一些浏览器里可能无法工作。

HTML:

```
<div id="food"><span class="dairy">milk</span><span class="dairy aged">cheese</span></div
```

Java：

```
WebElement cheese = driver.findElement(By.cssSelector("#food span.dairy.aged"));
```

By XPATH

此处略过不译

用户输入 - 填充表单

我们已经了解了怎么在输入框或者文本框中输入文字，但是如何操作其他的表单元素呢？你可以切换多选框的选中状态，你可以通过“点击”以选中一个 select 的选项。操作 select 元素不是一件很难的事情：

```
WebElement select = driver.findElement(By.tagName("select"));
List<WebElement> allOptions = select.findElements(By.tagName("option"));
for (WebElement option : allOptions) {
    System.out.println(String.format("Value is: %s", option.getAttribute("value")));
    option.click();
}
```

上述代码将找到页面中第一个 select 元素，然后遍历其中的每个 option，打印其值，再依次进行点击操作以选中这个 option。这并不是处理 select 元素最高效的方式。WebDriver 有一个叫 “Select” 的类，这个类提供了很多有用的方法用于 select 元素进行交互。

```
Select select = new Select(driver.findElement(By.tagName("select")));
select.deselectAll();
select.selectByVisibleText("Edam");
```

上述代码取消页面上第一个 select 元素的所有 option 的选中状态，然后选中字面值为 “Edam” 的 option。

如果你已经完成表单填充，你可能希望提交它，你只要找到 “submit” 按钮然后点击它即可。

```
driver.findElement(By.id("submit")).click();
```

或者，你可以调用 WebDriver 为每个元素提供的 “submit” 方法。如果你对一个 form 元素调用该方法，WebDriver 将调用这个 form 的 submit 方法。如果这个元素不是一个 form，将抛出一个异常。

```
element.submit();
```

在窗口和帧(frames)之间切换

有些 web 应用含有多个帧或者窗口。WebDriver 支持通过使用 “switchTo” 方法在多个帧或者窗口之间切换。

```
driver.switchTo().window("windowName");
```

所有 dirver 上的方法调用均被解析为指向这个特定的窗口。但是我们如何知道这个窗口的名字？来看一个打开窗口的链接：

```
<a href="somewhere.html" target="windowName">Click here to open a new window</a>
```

你可以将 “window handle” 传递给 “switchTo().window()” 方法。因此，你可以通过如下方法遍历所有打开的窗口：

for (String handle : driver.getWindowHandles()) { driver.switchTo().window(handle); }

你也可以切换到指定帧：

```
driver.switchTo().frame("frameName");
```

你可以通过点分隔符来访问子帧，也可以通过索引号指定它，例如：

```
driver.switchTo().frame("frameName.0.child");
```

该方法将查找到名为 “frameName” 的帧的第一个子帧的名为 “child” 的子帧。所有帧的计算都会从 **top** 开始。

弹出框

由 Selenium 2.0 beta 1 开始，就内置了对弹出框的处理。如果你触发了一个弹出框，你可以通过如下方访问到它：

```
Alert alert = driver.switchTo().alert();
```

该方法将返回目前被打开的弹出框。通过这个返回对象，你可以访问、关闭、读取它的内容甚至在 prompt 中输入一些内容。这个接口可以胜任 alerts,comfirms 和 prompts 的处理。

导航：历史记录和位置

更早的时候，我们通过 “get” 方法来访问一个页面 (driver.get("http://www.example.com"))。正如你所见，WebDriver 有一些更小巧的、聚焦任务的接口，而 navigation 就是其中一个非常有用的任务。因为加载页面是一个非常基本的需求，实现该功能的方法取决于 WebDriver 暴露的接口。它等同于如下代码：

```
driver.navigate().to("http://www.example.com");
```

重申一下: “navigate().to()” 和 “get()” 做的事情是完全一样的。只是前者更易用。

“navigate” 接口暴露了访问浏览器历史记录的接口：

```
driver.navigate().forward();
driver.navigate().back();
```

需要注意的是，该功能的表现完全依赖于你所使用的浏览器。如果你习惯了一种浏览器，那么在另一种浏览器中使用它时，完全可能发生一些意外的事情。

Cookies

在我们继续介绍更多内容之前，还有必要介绍一下如何操作 cookie。首先，你必须在 cookie 所在的域。如果你希望在加载一个大页面之前重设 cookie，你可以先访问站点中一个较小的页面，典型的是 404 页面 (http://example.com/some404page)。

```
// 进到正确的域
driver.get("http://www.example.com");

// 设置 cookie，这个cookie 对整个域都有效
Cookie cookie = new Cookie("key", "value");
driver.manage().addCookie(cookie);

// 输出当前 url 所有可用的 cookie
Set<Cookie> allCookies = driver.manage().getCookies();
for (Cookie loadedCookie : allCookies) {
    System.out.println(String.format("%s -> %s", loadedCookie.getName(), loadedCookie.get
}

// 你可以通过3中方式删除 cookie
// By name
driver.manage().deleteCookieNamed("CookieName");
// By Cookie
driver.manage().deleteCookie(loadedCookie);
// Or all of them
driver.manage().deleteAllCookies();
```

改变 UA

当使用 Firefox Driver 的时候这很容易：

```
FirefoxProfile profile = new FirefoxProfile();
profile.addAdditionalPreference("general.useragent.override", "some UA string");
WebDriver driver = new FirefoxDriver(profile);
```

拖拽

以下代码演示了如何使用 “Actions” 类来实现拖拽。浏览器本地方法必须要启用：

```
WebElement element = driver.findElement(By.name("source"));
WebElement target = driver.findElement(By.name("target"));

(new Actions(driver)).dragAndDrop(element, target).perform();
```

Driver 特性和权衡

Selenium-WebDriver’s Drivers

WebDriver 是编写测试时需要用到的方法的主要接口，这套接口有几套实现。包括：

HtmlUnit Driver

这是目前 WebDriver 最快速最轻量的实现。顾名思义，它是基于 HtmlUnit 的。HtmlUnit 是一个由 Java 实现的没有 GUI 的浏览器。任何非 Java 的语言绑定， Selenium Server 都需要使用这个 driver。

使用

```
WebDriver driver = new HtmlUnitDriver();
```

优势

- WebDriver 最快速的实现
- 纯 Java 实现，跨平台
- 支持 JavaScript

劣势

- 需要模拟浏览器中 JavaScript 的行为（如下）。

JavaScript in the HtmlUnit Driver

没有任何一个主流浏览器支持 HtmlUnit 使用的 JavaScript 引擎（Rhino）。如果你使用 HtmlUnit，测试结果可能和真实在浏览器中跑的很不一样。

当我们说到 “JavaScript” 时通常是指 “JavaScript 和 DOM”。虽然 DOM 由 W3C 组织定义，但是每个浏览器在 DOM 和 JavaScript 的交互的实现方面都有一些怪异和不同的地方。HtmlUnit 完全实现了 DOM 规范，并且对 JavaScript 提供了良好的支持，但它的实现和真实的浏览器都不一样：虽然它模拟了浏览器中的实现，但既不同于 W3C 指定的标准，也不同于其他主流浏览器的实现。

使用 WebDriver，我们需要做出选择：如果我们启用 HtmlUnit 的 JavaScript 支持，团队可能会遇到只有在这中情况下才会遇到的问题；如果我们禁用 JavaScript，但实际上越来越多的网站都依赖于 JavaScript。我们使用了最保守的方式，默认禁用 JavaScript 支持。对于 WebDriver 和 HtmlUnit 的每个发布版本，我们都会再次评估：这个版本是否可以默认开启 JavaScript 支持。

启用 **JavaScript**

启用 JavaScript 也非常简单：

```
HtmlUnitDriver driver = new HtmlUnitDriver(true);
```

上述代码会使得 HtmlUnit Driver 模拟 Firefox3.6 对 JavaScript 的处理。

Firefox Driver

我们通过一个 Firefox 的插件来控制 Firefox 浏览器。使用的配置文件是从默认安装的版本精简成只包含 Selenium WebDriver.xpi (插件) 的版本。我们还修改了一些默认配置（see the source to see which ones）,使得 Firefox Driver 可以运行和测试在 Windows、Mac、Linux 上。

使用

```
WebDriver driver = new FirefoxDriver();
```

优势

- 在真实的浏览器里运行，且支持 JavaScript
- 比 IE Driver 快

劣势

- 比 HtmlUnit Driver 慢
- 需要修改 Firefox 配置

例如你想修改 UA，但是你得到的是一个假的包含很多扩展的配置文件。这里有两种方式可以拿到真是的配置，假定配置文件是由 Firefox 配置管理器生成的：

```
ProfilesIni allProfiles = new ProfilesIni();
FirefoxProfile profile = allProfiles.getProfile("WebDriver");
profile.setPreferences("foo.bar", 23);
WebDriver driver = new FirefoxDriver(profile);
```

如果配置文件没有注册至 Firefox：

```
File profileDir = new File("path/to/top/level/of/profile");
FirefoxProfile profile = new FirefoxProfile(profileDir);
profile.addAdditionalPreferences(extraPrefs);
WebDriver driver = new FirefoxDriver(profile);
```

当我们开发 Firefox Driver 的特性时，需要评估它们是否可用。例如，直到我们认为本地方法在 Linux 的 Firefox 上是稳定的了，否则我们会默认禁用它。如需开启：

```
FirefoxProfile profile = new FirefoxProfile();
profile.setEnableNativeEvents(true);
WebDriver driver = new FirefoxDriver(profile);
```

信息

查看 Firefox section in the wiki page 以获得更多新鲜信息。

Internet Explorer Driver

这个 driver 由一个 .dll 文件控制，并且只在 windows 系统中可用。每个 Selenium 的发布版本都包含可用于测试的核心功能，兼容 XP 上的 ie6、7、8 和 Windows7 上的 ie9。

使用

```
WebDriver driver = new InternetExplorerDriver();
```

优势

- 运行在真实的浏览器中，并且支持 JavaScript，包括最终用户会碰到的一些怪异的问题。

劣势

- 显然它只在 Windows 系统上有效。
- 相对较慢。
- Xpath 在很多版本中都是非原生支持。Sizzle 会注入到浏览器，这使得它比其他浏览器要慢很多，也比在相同的浏览器中使用 CSS 选择器要慢。
- IE 6、7 不支持 CSS 选择器，由 Sizzle 注入替代。
- IE 8、9 虽然原生支持 CSS 选择器，但它们不完全支持 CSS3.

信息

访问 Internet Explorer section of the wiki page 以获得更多新鲜信息。特别注意配置部分的内容。

Chrome Driver

Chrome Driver 由 Chromium 项目团队自己维护和支持。WebDriver 通过 chromedriver 二进制包（可以在 chromiun 的下载页面找到）来工作。你需要确保同时安装了某版本的 chrome 浏览器和 chromedriver。chromedriver 需要存放在某个指定的路径下使得 WebDriver 可以自动发现它。chromedriver 可以发现安装在默认路径下的 chrome 浏览器。这些都可以被环境变量覆盖。请查看 wiki 以获得更多信息。

使用

```
WebDriver driver = new ChromeDriver();
```

优势

- 运行在真实的浏览器中，并且支持 JavaScript。
- 由于 chorme 是一个 webkit 内核的浏览器，Chrome Driver 能让你的站点在 Safari 中运行。注意自从 Chrome 使用了自己的 Javascript 引擎 V8 以后（之前是 Safari 的 Nitro 引

擎），Javascript 的执行可能会一点不一样。

劣势

- 比 HtmlUnit 慢

信息

查看 wiki 以获得更多最新信息。更多信息可以在 下载页面 找到。

运行 **Chrome Driver**

下载 Chrome Driver 并参考 wiki 上的其他建议。

Opera Driver

查看 wiki

iPhone Driver

查看 wiki

Android Driver

查看 wiki

可选择的后端：混合 WebDriver 和 RC 技术

WebDriver-Backed Selenium-RC

Java 版本的 WebDriver 提供了一套 Selenium-RC API 的实现。这意味着你可以使用 WebDriver 技术底层的 Selenium-RC API。这从根本上提供了向后兼容。这使得那些使用了 Selenium-RC API 的测试套件可以使用 WebDriver。这缓和了到 WebDriver 的迁移成本。同时，也允许你在同一个测试中使用两者的 API。

Selenium-WebDriver 的用法如下：

```
// 你可以使用任何 WebDriver 的实现，这里以 Firefox 的为例。
WebDriver driver = new FirefoxDriver();

// 基准 url，selenium 用于解析相对路径。
 String baseUrl = "http://www.google.com";

// 创建一个 Selenium 实现。
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);

// 使用 selenium 进行一些操作。
selenium.open("http://www.google.com");
selenium.type("name=q", "cheese");
selenium.click("name=btnG");

// Get the underlying WebDriver implementation back. This will refer to the
// same WebDriver instance as the "driver" variable above.
WebDriver driverInstance = ((WebDriverBackedSelenium) selenium).getWrappedDriver();

// 最后，通过调用 WebDriverBackedSelenium 实例的 stop 方法关闭浏览器。
// 应该避免使用 quit 方法，因为这样，在浏览器关闭后 jvm 还会继续运行。
selenium.stop();
```

优势

- 允许 WebDriver 和 Selenium API 并存。
- 提供了简单的机制从 Selenium RC API 迁移至 WebDriver。
- 不需要运行 Selenium RC server。

劣势

- 没有实现所有的方法。
- 一些高级用法可能无效（例如 Selenium Core 中的 “browserbot” 或其他内置的 js 方法）。
- 由于底层的实现，有些方法会比较慢。

Backing WebDriver with Selenium

WebDriver 支持的浏览器数量没有 Selenium RC 多，所以如果希望使用 WebDriver 时获得更多的浏览器支持，你可以使用 SeleneseCommandExecutor。

通过下面的代码，WebDriver 可以支持 safari（确保禁用弹出层）：

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setBrowserName("safari");
CommandExecutor executor = new SeleneseCommandExecutor(new URL("http://localhost:4444/"),
WebDriver driver = new RemoteWebDriver(executor, capabilities);
```

这种方案有一些明显的限制，特别是 findElements 不会如预期工作。同时，我们使用了 Selenium Core 来驱动浏览器，所以你也会受到 JavaScript 的沙箱限制。

运行 Selenium Server 以使用 RemoteDrivers¶

从 Selenium 下载页面 下载 selenium-server-standalone-.jar，你也可以选择下载 IEDriverServer。如果你需要测试 chrome，则从 google code 下载它。

把 IEDriverServer 和 chromedriver 解压到某个路径，并确保这个路径在 $PATH / %PATH% 中，这样 Selenium Server 就可以不需要任何设置就能操作 IE 和 chrome。

从命令行启动服务：

```
java -jar <path_to>/selenium-server-standalone-<version>.jar
```

如果你希望使用本地事件功能，在命令行添加以下参数：

```
-Dwebdriver.enable.native.events=1
```

查看帮助：

```
java -jar <path_to>/selenium-server-standalone-<version>.jar -help
```

为了运转正常，以下端口应该允许 TCP 请求链接：4444， 7054-5（或两倍于你计划并发运行的实例数量）。在 Windows 中，你可能需要 unblock 这个应用。

更多资源

你可以在 WebDriver wiki 找到更多有用的资源。

当然，你可以在互联网上搜索到更多 Selenium 的话题，包括 Selenium-WebDriver's drivers。有不少博客和众多论坛的帖子谈及到 Selenium。另外，Selenium 用户群组也是很重要的资源：http://groups.google.com/group/selenium-users。

接下来

本章节简要地从较高的层面介绍了 WebDriver 和其可信功能。一旦你熟悉了 Selenium WebDriver API 你可能会想要学习如何创建一个易于维护、可扩展的测试套件，并且提高哪些特性频繁修改的 AUT 的健壮性。大多数 Selenium 专家推荐的一种方式是：使用页面对象设计模式（可能是一个页面工厂）来设计你的测试代码。 Selenium WebDriver 在 Java 和 C sharp 中通过一个 PageFactory 类提供了这项支持。它同其他高级话题一样，将在下一章节讨论。同时，对于此项技术的较高层次的描述，你可以希望查看“测试设计考虑”章节。这两个章节都描述了如何通过模块化的思想使你的测试代码更易于维护。

Selenium 1 (Selenium RC)¶

介绍

正如你在 Selenium 项目简史里读到的，Selenium RC 在很长一段时间内都是 Selenium 的主要项目，直到 WebDriver/Selenium 合并而产生了最新和最强大的 Selenium 2。

Selenium 仍然被活跃的支持（大部分是维护工作），并且提供了一些 Selenium 2 短期不会支持的特性，包括支持多语言 (Java, Javascript, Ruby, PHP, Python, Perl 和 C#) 和支持几乎所有的浏览器。

Selenium RC 如何工作

首先，我们将讲述 Selenium RC 的组件如何操作，以及在测试脚本运行时各自扮演的角色。

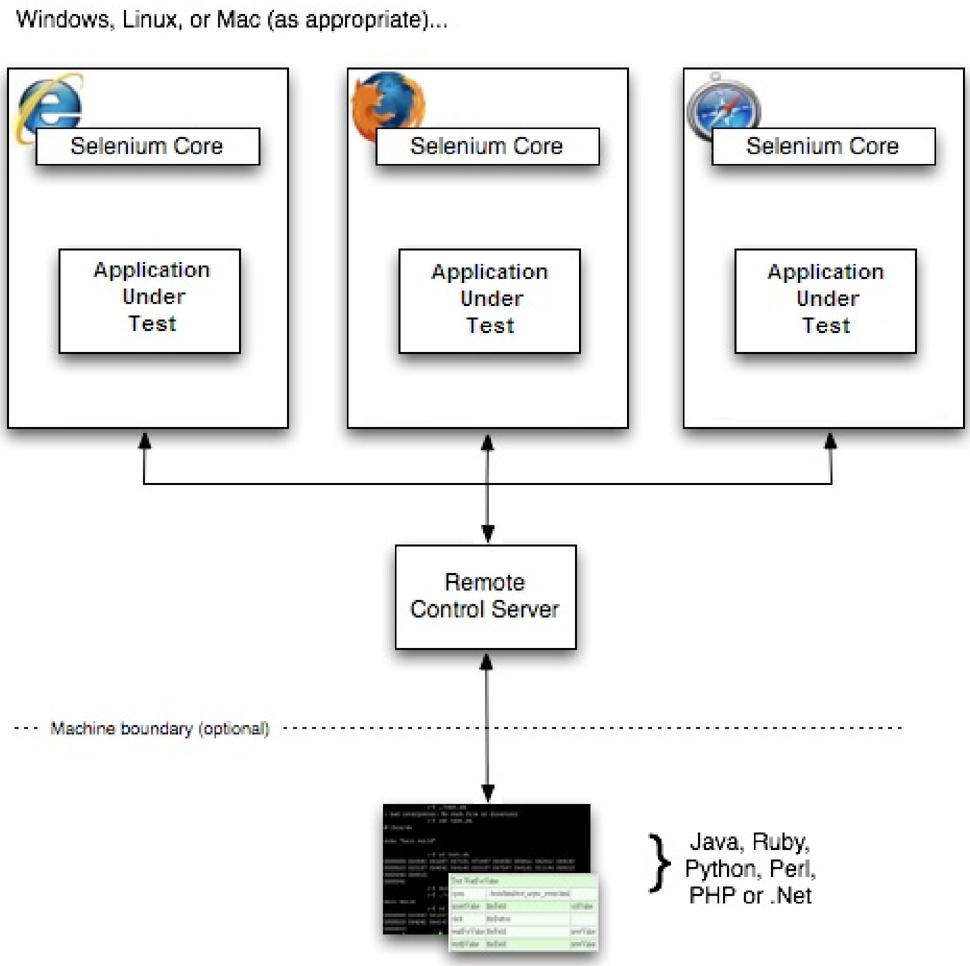
RC 组件

Selenium RC 组件是：

Selenium Server 能启动和杀死浏览器进程，解析并运行由测试程序传递过来的 Selenese 命令，并且可以是一个 HTTP 代理，拦截和验证浏览器和 AUT(测试中的应用)之间的 HTTP 通信。

客户端库提供了各种编程语言和 Selenium RC Server 之间的接口。

以下是一个简单的架构图：



上图演示了客户端和服务端进行通信以传递要执行的 Selenium 命令。然后服务端使用 Selenium-Core JavaScript 命令将 Selenium 命令传递给浏览器。浏览器则使用其内置的 JavaScript 解析器来执行 Selenium 命令。这样运行 Seleniun 动作或者验证你指定的测试脚本。

Selenium 服务端

Selenium 服务端从你的测试程序接收 Selenium 命令，解析它们，并且反馈给你程序的测试执行结果。

RC 服务端绑定了 Selenium Core 并且自动将其注入浏览器。这在你的测试程序打开浏览器时发生（使用客户端库的方法）。Selenium-Core 是一个 JavaScript 程序，实际上是一些利用浏览器的内置 JavaScript 解析器解析和实行 Selenese 命令 的 JavaScript 函数。

Server 使用简单的 HTTP GET/POST 请求来接收你的测试程序中的 Selenese 命令。这意味这你可以使用任何可以发送 HTTP 请求的编程语言来实现 Selenium 测试在浏览器中的自动运行。

客户端库

客户端库提供了能让你从自定义的程序中运行 Selenium 命令的编程支持。每种支持的语言都有一个不同的客户端库。Selenium 客户端库提供了一组接口，例如一些从你的程序中运行 Selenium 命令的方法。通过实现这些接口，我们就能得到一个支持所有 Selenese 命令的编程方法。

客户端库将 Selenese 命令传递给 Selenium 服务端来处理一个特定的动作或者执行 AUT 的测试。客户端库同时接收所传递命令的执行结果，并将其返回给你的程序。你的程序可以接收这个结果并且将其存储到一个变量中，然后报告其运行结果是成功还是失败，或者当其发生错误是进行适当的处理。

因此要创建一个测试程序，你仅仅需要使用客户端库的 API 来编写一个可以运行 Selenium 命令的程序。或者，如果你已经有了使用 Selenium-IDE 创建的 Selenium 测试脚本，你可以使用它来生成 Selenium RC 代码。Selenium-IDE 可以将 Selenium 命令转换（使用导出菜单）成客户端 API 的方法调用。查看 Selenium-IDE 章节中关于从 Selenium-IDE 中导出 RC 代码的细节。

安装

用安装这个词不是很恰当。Selenium 在你选择的编程语言中有一组组件可用。你可以从下载页面下载它们。

一旦你选定了一种编程语言，你仅需要：

- 安装 Selenium RC 服务端。
- 使用特定于该语言的客户端驱动创建你的项目

安装 Selenium 服务端

Selenium RC 服务端是一个简单的 jar 包 (selenium-server-standalone-.jar)，它不需要安装。只需要下载这个zip文件，并提取服务所需的目录即可。

运行 Selenium 服务

在开始任何测试之前，你必须先启动服务。进到 Selenium RC 服务端所在的目录，并在命令行中运行以下命令：

```
java -jar selenium-server-standalone-<version-number>.jar
```

你也可以简单的创建一个包含上述命令的批处理或shell文件（Windows 中扩展名为 .bat，Linux 中扩展名为 .sh）。然后在你的桌面上创建一个该可执行文件的快捷方式，通过双击图标来启动服务。

要成功启动服务必须确保 Java 已安装，并且设置了正确的 PATH 环境变量。你可以通过下面的命令检查你的 Java 是否安装正确：

```
java -version
```

如果你得到一个版本号（必须>=1.5），那么你已经成功启动 Selenium RC。

使用 Java 客户端驱动

- 从 SeleniumHQ 下载页面下载 Selenium java 客户端驱动 zip 包。
- 提取 selenium-java-.jar
- 打开你喜欢的 Java IDE (Eclipse, NetBeans, IntelliJ, Netweaver, etc.)
- 创建一个 java 项目。
- 将 selenium-java-.jar 文件作为引用添加到你的项目中。
- 将 selenium-java-.jar 文件添加到你项目的 classpath 中。
- 从 Selenium-IDE 到处一个 Java 文件，并放入你的项目，或者使用 Selenium 的 Java 客户端 API 编写一个 Selenium 测试文件。这些 API 将在本章的后面部分进行讲解。你可以使用 JUnit，或者 TestNg 来运行你的测试，或者你可以简单的写一个 main() 方法。这些概念也将在本文后面进行说明。
- 从命令行运行 Selenium 服务。
- 从 Java IDE 或者命令行中执行你的测试。

关于更多 Java 测试项目的配置细节，可查看本章附件：在 **Eclipse** 中配置 **Selenium RC** 和在 **Intellij** 中配置 **Selenium RC**。

将 Selenese 转换成程序

使用 Selenium RC 的主要任务就是将你的 Selenese 转换成一个编程语言。在本小结中，我们提供几种不同的语言演示。

测试脚本范例

让我们从一个 Selenese 测试脚本的例子开始. 假定我们使用 Selenium-IDE 记录了如下测试：

open	/	
type	q	selenium rc
clickAndWait	btnG	
assertTextPresent	Results * for selenium rc	

注意: 这个例子仅仅在 Google 搜索页面 http://www.google.com 工作。

Selenese 作为编程代码

以下为使用支持的多种编程序言从 Selenium-IDE 中导出的测试脚本。如果你有一些面向对象编程的基础知识，你就可以通过阅读以下代码理解 Selenium 如何运行 Selenese 命令。

```
/** Add JUnit framework to your classpath if not already there
 *  for this example to work
 */
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class NewTest extends SeleneseTestCase {
    public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
    }
      public void testNew() throws Exception {
          selenium.open("/");
          selenium.type("q", "selenium rc");
          selenium.click("btnG");
          selenium.waitForPageToLoad("30000");
          assertTrue(selenium.isTextPresent("Results * for selenium rc"));
    }
}
```

在接下来的章节中，我们将介绍如何通过生成的代码创建你的测试程序。

编写你的测试代码

现在我们将为每种支持的语言演示如何通过上述例子编写你自己的测试代码。我们主要需要做2件事情：

- 从 Selenium-IDE 导出指定语言的脚本，有选择性的修改它。
- 编写一个 main() 方法来执行创建的代码。

你可以选择平台支持的任意测试引擎，如 Java 的 JUnit 或 TestNG。

这里我们将演示指定语言的例子。每种语言的 API 都有所不同，所以我们将单独解释每一个。

Java

在 Java 中，大家通常选择 JUnit 或 TestNG 作为测试引擎。一些像 Eclipse 这样的 IDE 能通过插件直接支持它们，使得事情更简单。JUnit 和 TestNG 教学不在本文档的范围内，但是你可以通过网络找到相关资料。如果你是一个 Java 程序员，你可能已经有使用这些框架的经验了。

你可能希望为 “NewTest” 测试类重命名。同时，你可能也需要修改以下语句中的浏览器打开参数。

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.google.com/");
```

使用 Selenium-IDE 创建的代码看起来大致如下。为了使代码更清晰易读，我们手工加入了注释。

```
package com.example.tests;
// 我们指定了这个文件的包

import com.thoughtworks.selenium.*;
// 导入驱动。
// 你将使用它来初始化浏览器并执行一些任务。

import java.util.regex.Pattern;
// 加入正则表达式模块，因为有些我们需要使用它进行校验。
// 如果你的代码不需要它，完全可以移除掉。

public class NewTest extends SeleneseTestCase {
// 创建 Selenium 测试用例

      public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
             // 初始化并启动浏览器
      }

      public void testNew() throws Exception {
           selenium.open("/");
           selenium.type("q", "selenium rc");
           selenium.click("btnG");
           selenium.waitForPageToLoad("30000");
           assertTrue(selenium.isTextPresent("Results * for selenium rc"));
           // 以上为真实的测试步骤
     }
}
```

学习使用 API

Selenium RC API 使用以下约定：假设你了解 Selenese，并且大部分接口是自解释的。在此，我们仅解释最具争议或者看起来不那么直接明了的部分。

启动浏览器

```
setUp("http://www.google.com/", "*firefox");
```

每个例子都打开了一个浏览器，并且将浏览器作为一个浏览器对象返回，赋值给一个变量。这个变量将用于调用浏览器方法。这些方法可以执行 Selenium 命令，例如打开、键入或者校验。

创建浏览器对象所需要的参数如下：

host

指定服务所在的机器的 IP 地址。通常它和运行客户端的机器是同一台。所以在这个例子中我们传入 localhost。在某些客户端中，这是一个可选参数。

port

指定服务监听的客户端用于创建连接的 TCP/IP socket。这在某些客户端中也是可选的。

browser

指定你希望运行测试的浏览器。该参数必选。

url

AUT 的基准 url。在所有的客户端中必选，并且是启动浏览器代理的 AUT 通讯的必须信息。

注意，有些客户端要求调用 start() 方法来启动浏览器。

运行 命令

一旦你初始化了一个浏览器并且将其赋值给一个变量（通常命名为 "Selenium"），你可以使用这个变量调用各种方法来运行 Selenese 命令。例如，调用 selenium 对象的键入方法：

```
selenium.type(“field-id”,”string to type”)
```

此时浏览器将真正执行指定的操作，在这个方法调用时指定了定位符和要键入的字符串，本质上就像是一个用户在浏览器中输入了这些内容。

报告结果

Selenium RC 没有内置的结果报告机制。而是让你根据所选语言的特性创建符合你需求的自定义报告。这非常棒！但是你是不是希望这些事情都已经就绪，而你可以快速使用它们？其实市面上不难找到符合你需求的库或框架，这比编写你自己的测试报告代码快多了。

测试框架报告工具

很多语言都有对应的测试框架。它们除了提供灵活的测试引擎执行你的测试之外，通常还包括结果报告的库。例如，Java有两个常用的测试框架，JUnit 和 TestNG. .NET 也有适合它的, NUnit。

我们不会教你如何使用这些框架，那超出了本指南的范围。但我们将简单介绍一下这些框架中你可以使用的跟 Selenium 相关的特性。有很多关于学习这些测试框架的书，互联网上页有丰富的资料。

测试报告库

同样可以利用的是使用你所选语言编写的专门用于报告测试结果的三方库。它们通常支持多种格式，如 HTML 或 PDF。

最佳实践是？

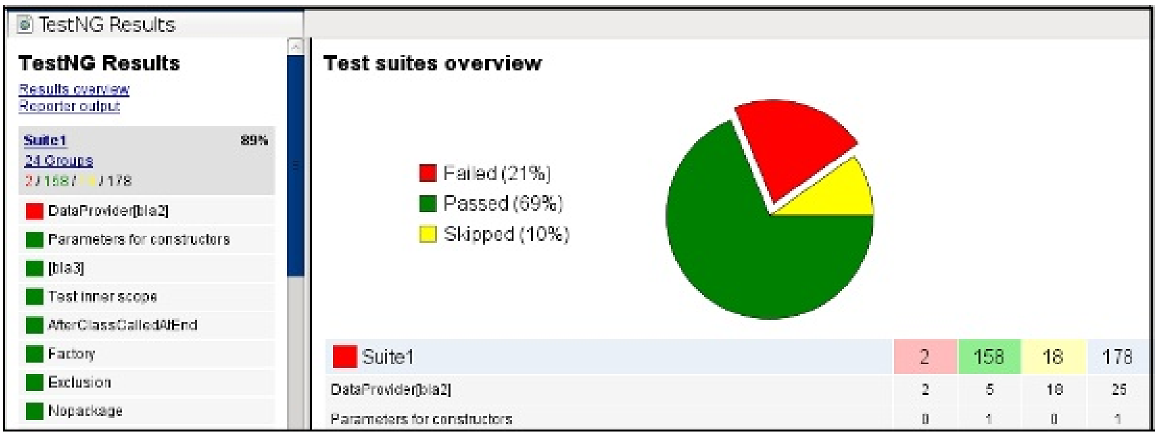
大多数新接触测试框架的人将会从框架内置的报告功能开始。他们会检查任何可用库，这可比你自己开发的开销要小。当你开始使用 Selenium，毫无疑问你将开始在报告处理中使用你自己的 “print 语句”。这将可能导致你在使用一个库或框架的同时，逐渐开发开发你自己的报告功能。无论如何，在最初短暂的学习曲线之后，你将自然而然的开发出最适合你的报告功能。

测试报告范例

为了进行演示，我们将直接使用 Selenium 支持的语言的特定工具。以下列出的是最常用的，而且也是最为推荐的。

Java 中的测试报告

- 如果 Selenium 测试用例是使用 JUnit 开发的，那么 JUnit 报告就能用于创建测试报告。了解更多 JUnit 报告 。
- 如果 Selenium 测试用例是使用 TestNG 开发的，那也不需要依赖外部任务来创建测试报告。TestNG 框架创建包含测试详情列表的 HTML 报告。了解更多 TestNG 报告 。
- ReportNG 是一个用于TestNG 框架的 HTML 报告插件。它的初衷是用于取代默认的 HTML 报告。ReportNG 提供了简单、彩色的测试结果显示。了解更多 TestNG
- 同时，TestNG-xslt 是一个很好的摘要报告工具。TestNG-xslt 报告看起来如下图：



了解更多 TestNG-xslt

记录 Selenese 命令

Logging Selenium 可以用于为你的测试创建一个含有所有 Selenium 命令及其运行结果（成功或失败）的报告。为了获得这项功能，使用 Logging Selenium 扩展你的 Java 客户端。了解更多 Logging Selenium

为你的测试加点料

现在我们将获得所有使用 Selenium 的理由，它能为你的测试添加逻辑。就像任何程序一样。程序流通过条件语句和迭代控制。另外，你能使用 IO 来报告处理信息。在这一小结中，我们将演示一些可联合 Selenium 使用的编程语言构建例子，用以解决常见的测试问题。

当你将页面元素是否存在的简单测试转换成涉及多个网页和数据的动态功能时，你将发现你需要编程逻辑来校验期待的结果。一般的， Selenium-IDE 不支持迭代和标准的条件语句。你可以通过将 javascript 嵌入 Selenese 参数来实现条件控制和迭代，并且大部分的条件都比真正的编程语言要简单。此外，你可能需要使用异常处理来进行错误回复。基于这些原因，我们编写了这一小结内容来演示普通编程技巧的使用，以使你在自动化测试中获得更大的校验能力。

本小结例子使用 C# 和 Java 编写而成，它们非常简单，也很容易转换成其他语言。如果你有一些面向对象编程的基础知识，你将很容易掌握这个章节。

迭代

迭代是测试中最常用的功能了。例如你可能希望执行一个查询多次。或者你需要处理那些从数据库中返回的结果集以校验你的测试结果。

使用同之前一样的 Google 搜索例子，让我们来检查搜索结果。这个测试将使用 Selenese：

open	/	
type	q	selenium rc
clickAndWait	btnG	
assertTextPresent	Results * for selenium rc	
type	q	selenium ide
clickAndWait	btnG	
assertTextPresent	Results * for selenium ide	
type	q	selenium grid
clickAndWait	btnG	
assertTextPresent	Results * for selenium grid	

同样的代码重复跑了3次。将同样的代码拷贝多次运行可不是一个好的编程实践，因为维护的时候成本会很高。使用编程语言，我们可以通过迭代这一更灵活更易于维护的方式来处理搜索结果。

In Csharp

```
// Collection of String values.
String[] arr = {"ide", "rc", "grid"};

// Execute loop for each String in array 'arr'.
foreach (String s in arr) {
    sel.open("/");
    sel.type("q", "selenium " +s);
    sel.click("btnG");
    sel.waitForPageToLoad("30000");
    assertTrue("Expected text: " +s+ " is missing on page."
    , sel.isTextPresent("Results * for selenium " + s));
}
```

条件语句

我们使用一个例子来演示条件语句的使用。让运行 Selenium 测试时，如果一个原本应该存在的元素没有出现在页面上时，将会触发一个普通的错误。例如，我们运行如下 代码：

```
// Java
selenium.type("q", "selenium " +s);
```

如果元素“q”不在页面上将会抛出一个异常：

```
com.thoughtworks.selenium.SeleniumException: ERROR: Element q not found
```

这个异常将会终止你的测试。对于某些测试来说这正是你想要的。但是更多的时候，你并不希望这样，因为还有很多后续的测试要执行。

一个更好的解决办法是我们首先判定元素是否存在，然后再进行相应的处理。我们来看看 Java 的写法：

```
// 如果元素可用，则则行类型判定操作
if(selenium.isElementPresent("q")) {
    selenium.type("q", "Selenium rc");
} else {
    System.out.printf("Element: " +q+ " is not available on page.")
}
```

这样做的好处是，即使页面上没有这个元素测试也能够继续执行。

在你的测试中执行 JavaScript

在一个应用程序中使用 JavaScript 是非常方便的，但是 Selenium 不直接支持它。你可以在 Selenium RC 中使用 getEval 接口的方法来执行它。

考虑一个应用中的没有静态 id 的多选框。在这种情况下，你可以通过使用 Selenium RC 对 JavaScript 语句进行求值（evaluate）来找到所有的多选框并处理它们。

```
// Java
public static String[] getAllCheckboxIds () {
     String script = "var inputId  = new Array();";// Create array in java script.
            script += "var cnt = 0;"; // Counter for check box ids.
            script += "var inputFields  = new Array();"; // Create array in java script.
            script += "inputFields = window.document.getElementsByTagName('input');"; //
            script += "for(var i=0; i<inputFields.length; i++) {"; // Loop through the co
            script += "if(inputFields[i].id !=null " +
                      "&& inputFields[i].id !='undefined' " +
                      "&& inputFields[i].getAttribute('type') == 'checkbox') {"; // If in
            script += "inputId[cnt]=inputFields[i].id ;" + // Save check box id to inputI
                      "cnt++;" + // increment the counter.
                      "}" + // end of if.
                      "}"; // end of for.
            script += "inputId.toString();" ;// Convert array in to string.
     String[] checkboxIds = selenium.getEval(script).split(","); // Split the string.
     return checkboxIds;
 }
```

如果要计算页面中的图片数，你可以：

```
// Java
selenium.getEval("window.document.images.length;");
```

记住要调用 window 对象，以防在 DOM 表达式中其默认指向 Selenium 窗口而不是测试窗口。

服务端选项

当服务启动时，可以使用命令行配置项来改变其默认行为。

回想一下，我们是这样启动服务的：

```
$ java -jar selenium-server-standalone-<version-number>.jar
```

你可以使用 -h 来查看所有的配置项：

```
$ java -jar selenium-server-standalone-<version-number>.jar -h
```

你将看到所有配置项列表，每个配置项附带间断描述。这里提供的描述并不总是足够禽畜，所以接下来我们将对一些重要的配置项进行补充描述。

代理配置

如果你的 AUAT 使用了一个需要授权的 HTTP 代理，你需要使用以下命令来配置 http.proxyHost, http.proxyPort, http.proxyUser 和 http.proxyPassword。

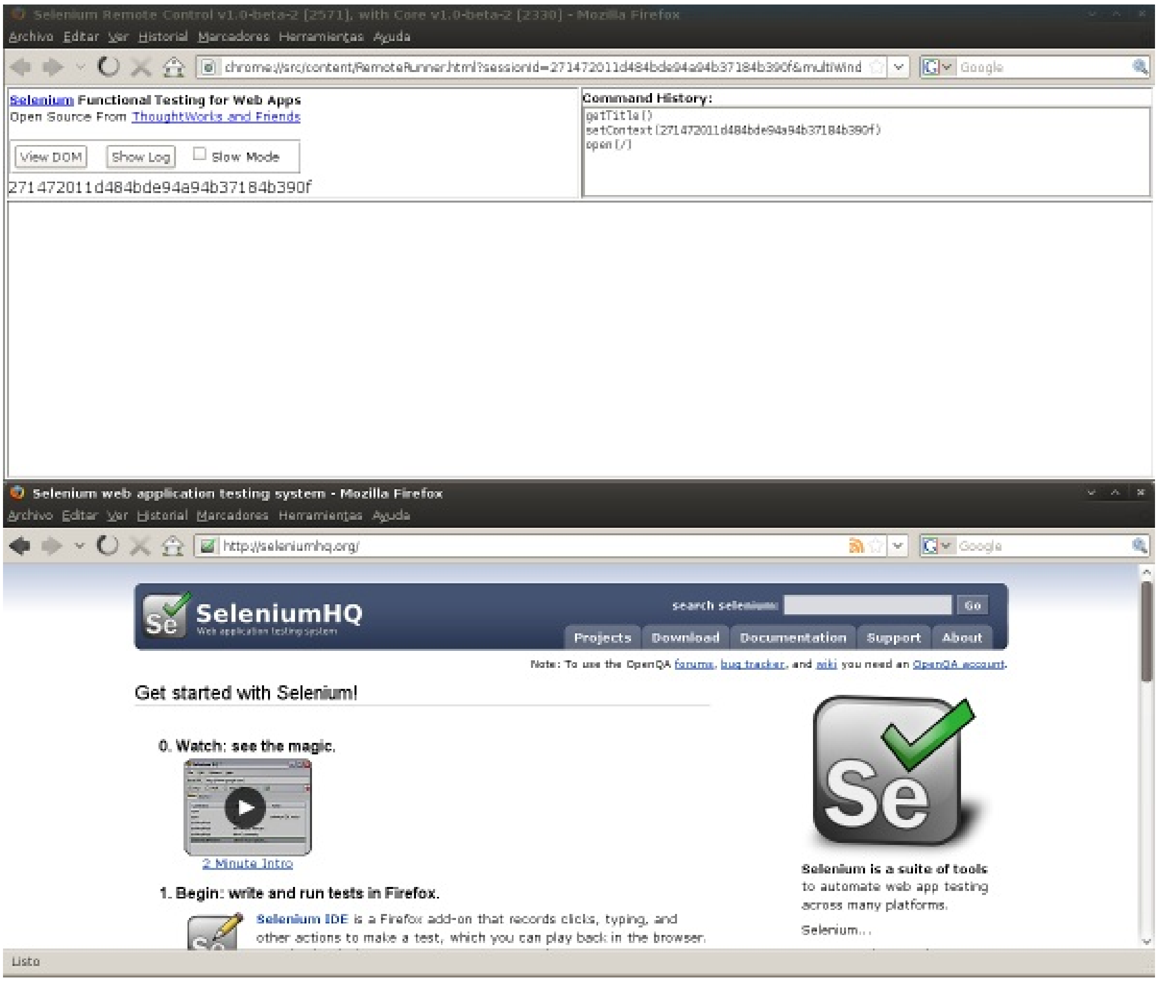
```
$ java -jar selenium-server-standalone-<version-number>.jar -Dhttp.proxyHost=proxy.com -D
```

多窗口模式

如果你正在使用 Selenium 1，你可以跳过这部分内容，因为多窗口模式已经是默认配置。但是在更早的版本中，AUT 默认是在子帧(sub frame)中运行的。



有些应用在子帧中不能正常运行，必须要加载到顶级帧中运行。多窗口模式允许 AUT 在两个独立的窗口中运行，而不是在默认的帧中运行，这样它就能在顶级帧中运行了。



对于老版本的 Selenium 来说，你必须通过下面的配置项明确指定多窗口模式：

```
-multiwindow
```

在 Selenium 1 以及更新的版本中，如果你希望在单窗口中运行你的测试，你可以使用以下配置项：

```
-singlewindow
```

指定 Firefox 配置

Firefox 不会同时运行两个实例，除非你为每一个指定单独的配置。Selenium RC 1 及其后续版本会自动运行两个单独的配置，所以如果你正在使用 Selenium 1，你可以跳过这个章节。如果你在使用更老的版本而你有需要指定单独的配置，你需要明确的指定它。

首先，穿加你一个单独的 Firefox 配置，根据以下步骤。打开 Windows 的开始菜单，选择 “run”，然后键入以下内容：

```
firefox.exe -profilemanager

firefox.exe -P
```

使用对话框来创建新配置。当你运行 Selenium 服务时，你需要使用命令行选项 -firefoxProfileTemplate 告诉它使用新的 Firefox 配置，并且指定要使用的配置的路径。

```
-firefoxProfileTemplate "path to the profile"
```

警告

确保你的配置文件被存放在一个不同于默认路径的文件夹中！！！Firefox 配置管理会在你删除一个配置的时候删除该配置所在文件夹的所有内容，而不管它是不是配置文件。

更多请参考 Mozilla’s Knowledge Base

通过 -htmlSuite 配置项在服务端直接运行 Selenese

通过将 html 文件传递给服务端的命令行，你可以直接在 Selenium 服务端运行 Selenese html 文件。例如：

```
java -jar selenium-server-standalone-<version-number>.jar -htmlSuite "*firefox"
"http://www.google.com" "c:\absolute\path\to\my\HTMLSuite.html"
"c:\absolute\path\to\my\results.html"
```

这个例子将自动加载你的 html 测试套件，运行所有的测试并生成一份 html 格式的测试报告。

注意

在使用这个配置项时，服务端将开始运行测试，并为测试结束等待指定的秒数，如果测试没有在指定时间内结束，命令行将以一个非0的退出码退出，并且没有报告文件生成。

这个命令行非常长，所以键入它的时候需要非常小心。注意这要求你传入一个 html 测试套件，而非单个的测试。并且配置项和 -interactive 不兼容，你不能同时使用他们。

Selenium 服务日志

服务端日志

当启动 Selenium 服务，可以使用 -log 配置项来将 Selenium 服务报告的有价值的 debug 信息记录到一个文本文件。

```
java -jar selenium-server-standalone-<version-number>.jar -log selenium.log
```

这个日志文件相比标准的 console 日志而言要冗余的多（它包括了 debug 级别的日志信息）。它页包含了 logger name，打印日志信息的线程 id。例如：

```
20:44:25 DEBUG [12] org.openqa.selenium.server.SeleniumDriverResourceHandler -
Browser 465828/:top frame1 posted START NEW
```

该信息格式为：

```
TIMESTAMP(HH:mm:ss) LEVEL [THREAD] LOGGER - MESSAGE
```

浏览器端日志

在浏览器端的 javascript （Selenium Core）也将记录重要的日志信息。在很多时候，对最终用户而言，这比常规的 Selenium 服务端日志有用的多。为了访问浏览器端日志，将 -browserSideLog 参数传递给 Selenium 服务。

```
java -jar selenium-server-standalone-<version-number>.jar -browserSideLog
```

为了将所有浏览器端的日志保存到一个文件中，-browserSideLog 必须和 -log 配置项联合使用。

指定特定浏览器路径

你可以为 Selenium RC 指定一个特定浏览器的路径。如果你需要测试同一个浏览器的不同版本时，这一功能将非常有效。同时这也允许你在一个 Selenium RC 不直接支持的浏览器中运行你的测试。当指定这个运行模式，使用 *cunstom 来指定可执行的浏览器的全路径：

```
*custom <path to browser>
```

Selenium RC 架构

注意

该主题尝试解释 Selenium RC 背后的运行原理。这并不是 Selnium 用户需要了解的基础知识，但是你会发现它对于了解一些问题非常有用。

为了理解 Selenium RC 服务端工作的细节，以及为什么它使用代理注入和高特权模式你必须先了解 同源策略。

同源策略

Selenium 面临的主要约束即同源策略。市面上所有的浏览器都有这个安全约束，它的目的是确保一个网站的内容永远不会被另外一个站点的脚本访问到。同源策略规定浏览器加载的任何脚本仅能操作引入它的页面所在的域的内容。它也不能执行另一个网站中的方法。例如，如果浏览器在载入 www.mysite.com 时加载了一个脚本，这脚本就不能操作

www.mysite2.com 的内容，即使那是另一个你自己的网站。如果这被允许，脚本将可以操作你打开的任何网站的内容，于是当你在 tab 页中打开一个银行站点时它就能读取你的银行账号信息。。我们把这叫 XSS(Cross-site Scripting) 攻击。

为了在这个约束下工作，Selenium Core（包括它的 javascript 脚本）必须和 AUT 放置在同一个域下。

之前，因为 Selenium core 使用 JavaScript 实现的，所以一直被这个问题困扰。但现在，这个问题已经得到解决。它使用 Selenium 服务端作为一个代理来避免这个问题。本质上来讲，Selenium RC 告诉浏览器它是运行在服务端提供的一个“被欺骗的”站点上。

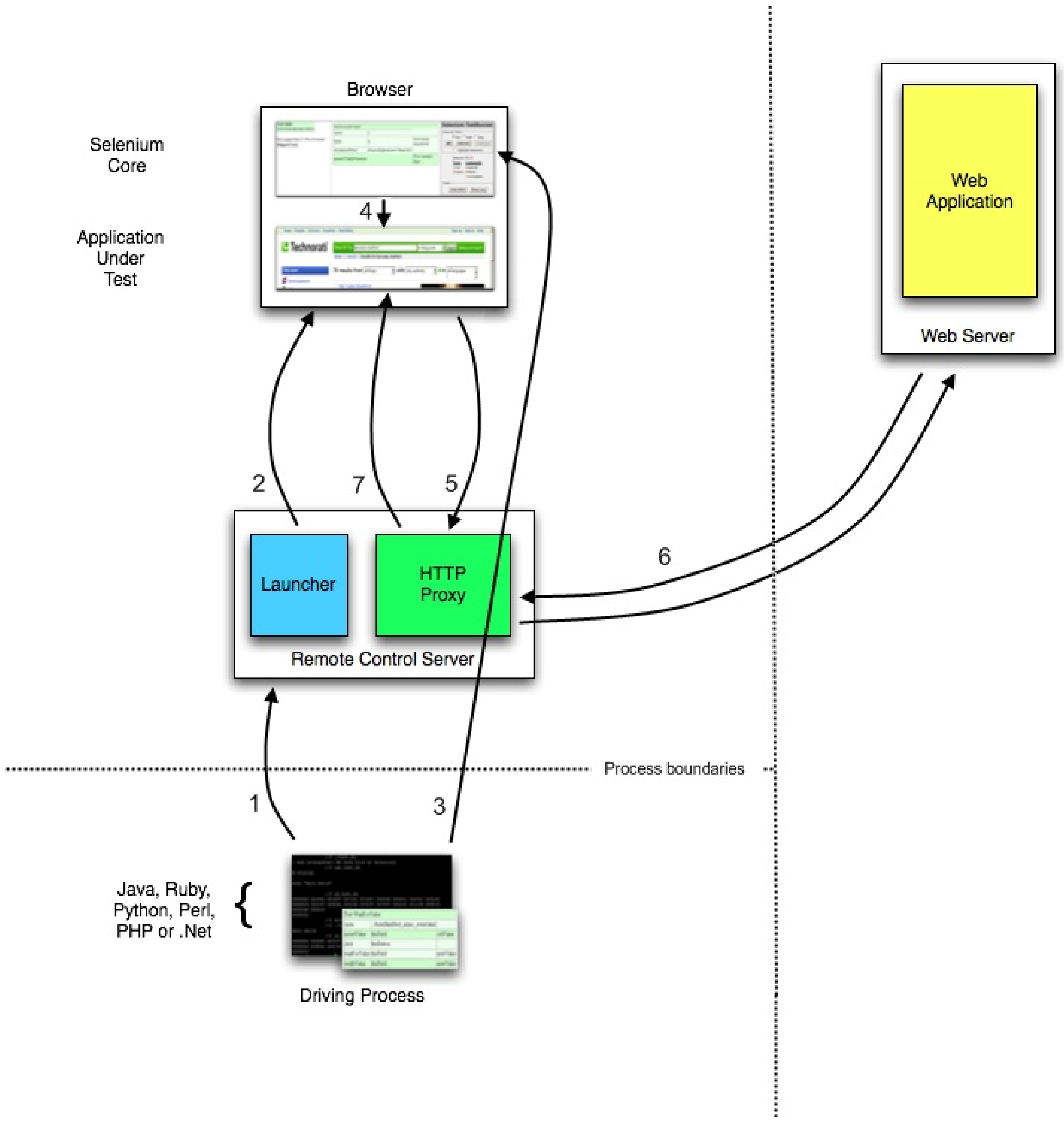
注意

你可以在维基百科上找到更多关于 同源策略 和 XSS 的内容

代理注入

Selenium 避免同源策略约束的首选方法是代理注入。在代理注入模式，Seleniium 服务端扮演一个客户端配置[1] 的 HTTP 代理[2] 的角色，它位于浏览器和 AUT 之间。它为 AUT 伪装了一个虚假的 url（将Selenium Core 和测试注入到 AUT，就好像他们来自同一个域）。

1. 代理扮演一个第三方角色，在双方传递内容的过程中。它好像一个 web 服务器将 AUT 传送给浏览器。作为一个代理，使得 Selenium 服务端有能力伪装 AUT 的真实 url。
2. 浏览器加载的时候，配置文件将指定 localhost:4444 作为 http 代理，这就是为什么浏览器发起一个 http 请求将通过 Selenium 服务端并且响应页将通过它而不是来自真实的服务器。以下是结构图：



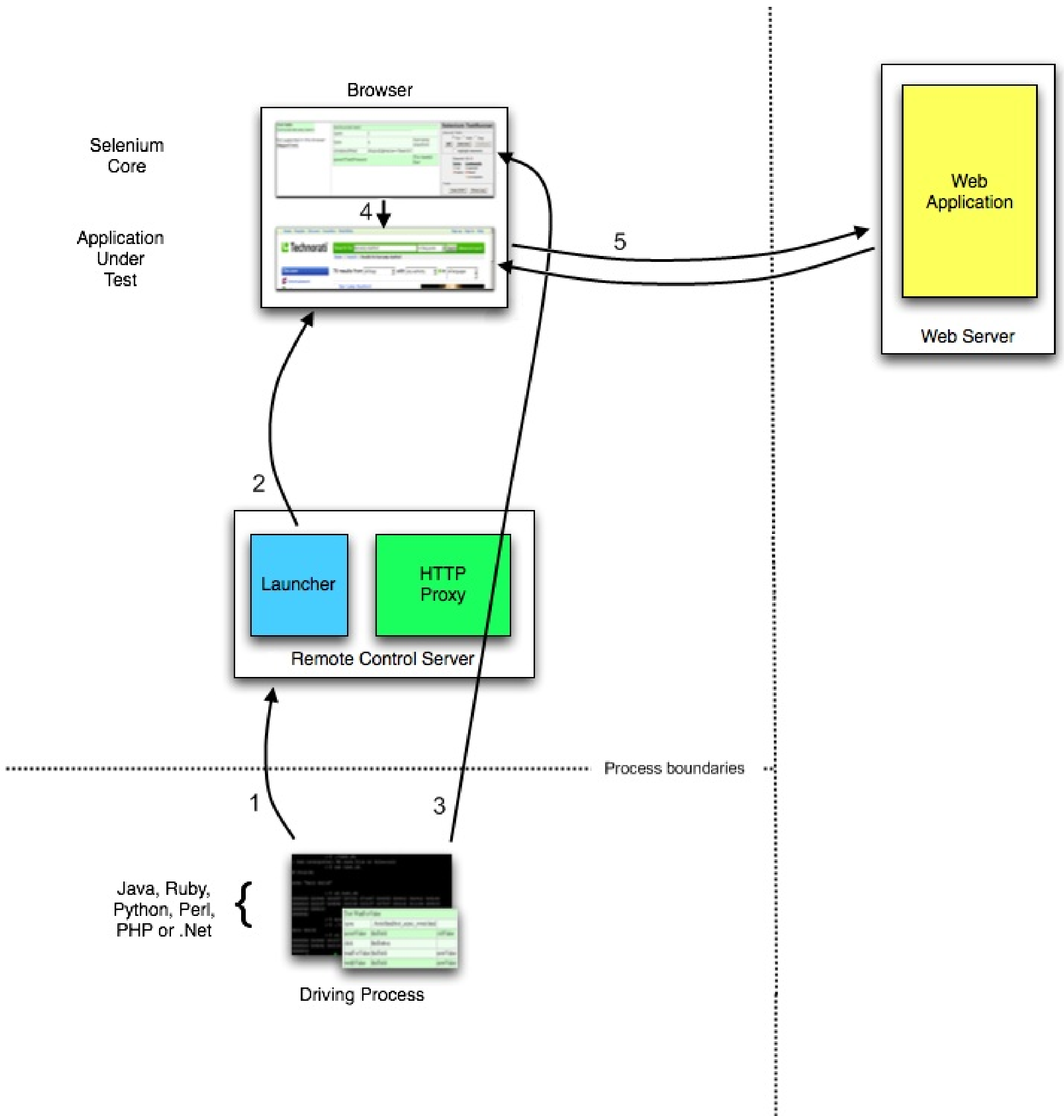
当测试开始时，将发生以下事情：

1. 客户端驱动将和 Selenium RC 服务端建立一个连接。
2. Selenium RC 服务端启动一个打开指定 url 的浏览器（或复用一个已打开的），将 Selenium Core 的 JavaScript 代码注入的这个页面中。
3. 客户端驱动向服务端传递一个 Selenese 命令。
4. 服务端解析这个命令，然后触发 JavaScript 脚本执行浏览器中相应的命令。
5. Selenium Core 指示浏览器在第一个指令后开始执行，典型的是打开一个 AUT 页面。
6. 浏览器收到打开页面的请求，并且从 Selenium RC 服务端询问获取页面内容（作为浏览器的 http 代理）
7. Selenium RC 服务端和网站服务器通讯，一旦获取到页面，它就对页面的源进行伪装然后发送到浏览器，使这个页面看起来像是和 Selenium Core 来自于同一个源（这使得我们可以绕开同源策略的限制）
8. 浏览器接收到这个页面并且渲染到相应的帧或者窗口。

高特权浏览器（Heightened Privileges Browsers）

这种方法的工作流程和代理注入非常像，主要的区别是浏览器在一个叫高特权的模式下启动，这将允许网站做一些平时不被允许做的事情（例如 XSS，或者填充文件上传输入框，或者其他一些对 Selenium 非常有用的操作）。使用这种浏览器模式， Selenium Core 就可以直接打开 AUT 并且读取或操作其内容，而不需要将整个 AUT 通过 Selenium RC 服务端中转。

结构图如下：



此时，将发生以下事情：

1. 客户端驱动和 Selenium RC 服务端建立一个连接。
2. Selenium RC 服务端启动一个开打指定 url 的浏览器，并且将 Selenium Core 加载到整个页面中。
3. Selenium Core 从客户端驱动获得第一个指令（通过向 Selenium RC 服务端发起的另一个 http 请求）。

4. Selenium Core 执行第一个指令，典型的是打开一个 AUT 页面。
5. 浏览器收到这个请求并且向站点服务器请求页面。一旦浏览器接收到页面内容，就会渲染到相应的帧或窗口。

处理 HTTPS 和安全警告弹出框

当需要发送诸如密码或信用卡等加密信息时，我们往往会从 http 转为 https。这在今天的应用中非常常见。Selenium RC 也支持。

为了确保这个 https 站点的真实性，浏览器需要一个安全整数。否则，当浏览器使用 https 访问 AUT 时，这个应用经常被认为是不受信任的。当遇到这种情况时，浏览器会显示安全警告弹出框，而 Selenium RC 无法关闭这个弹出框。

当在 Selenium RC 测试中使用 https 时，你必须使用一个支持的运行模式，并且能为你处理安全证书。你可以在测试项目初始化 Selenium 时指定这个运行模式。

在 Selenium RC 1.0 beta 2 和其后续版本中，可以使用 *firefox* 和 iexplore 运行模式。在更早期的版本中，包括 Selenium RC 1.0 beta 1 使用 *chrome* 和 iehta 运行模式。通过使用这些运行模式，你不需要安装任何特殊的安全证书，Selenium RC 将帮你处理它。

在版本1中，推荐运行 *firefox* 和 iexplore 运行模式。然而，我们还提供 *iexploreproxy* 和 firefoxproxy 运行模式。它们只是用于提供向后兼容，除非遗留的测试项目，否则我们不应该使用它们。在你需要处理安全证书和运行多窗口时，它们的处理将存在局限性。

在 Selenium RC 的早期版本中，*chrome* 或 iehta 是支持 https 和能处理安全警告弹出窗的运行模式。它们被认为是实验性的模式，虽然现在它们已经很稳定并且有大量用户。如果你在使用 Selenium 1，你不应该使用那些老的运行模式。

关于安全证书

通常来来说，安装了安全证书后，浏览器将信任你测试的应用。你可以在浏览器的选项或者 Internet 属性中检查它（如果你不知道你的 AUT 的安全证书，询问你的系统管理员）。当 Selenium 启动了浏览器，它注入代码以解析浏览器和服务器之间的通讯。这时，浏览器认为这个引用是不被信任的了，并且会弹出一个安全警告。

为了绕过这个问题，Selenium RC，（又需要使用支持的运行模式）将安装它自己的证书。将临时装在你的客户机上，能被浏览器访问到的地方。这将欺骗浏览器认为它在访问一个和你的 AUT 完全不同的重难点，就能成功的组织弹出框。

另一个在早期的版本中解决此问题的方法是安装一个随 Selenium 安装提供的 Cybervillians 安全证书。大部分用户不需要做这件事情，但是当你在代理注入的模式下运行 Selenium RC 时，你就需要安装它了。

更多浏览器支持和相关配置

Selenium API 支持在多个浏览器中运行，包括 ie 和 Firefox。请从 SeleniumHQ.org 查看支持的浏览器。另外，当一个浏览器不直接被支持时，启动浏览器时，你可以使用 ”*custom“* 来指定一个浏览器运行你的 *Selenium* 测试（例如：替换 firefox 或 *iexplore）。这样，你可以将这个 API 调用可执行的路径传递给浏览器。这个操作也可以在服务端的交互模式下完成。

```
cmd=getNewBrowserSession&1=*custom c:\Program Files\Mozilla Firefox\MyBrowser.exe&2=http:
```

使用不同的浏览器配置来运行测试

通常 Selenium RC 会自动配置浏览器, 但是如果你使用 “*custom” 运行模式启动浏览器，你必须强制 Selenium RC 启动浏览器，就像自动配置不存在一样。

例如，你使用如下自定义配置启动 Firefox：

```
cmd=getNewBrowserSession&1=*custom c:\Program Files\Mozilla Firefox\firefox.exe&2=http://
```

注意，当使用这种方法启动浏览器时，我们必须手工配置浏览器使用 Selenium 服务端作为代理。通常这意味这你需要打开你的浏览器选项，指定 “localhost:4444” 作为 http 代理，但是每种浏览器的设置方式可能不太一样。

注意 Mozilla 浏览器的启动和停止不太一样。你需要设置 MOZ_NO_REMOTE 环境变量确保它表现如预期。Unix 用户应该避免使用 shell 脚本来启动它，直接使用一个二进制可执行文（如：firefox-bin）会更好。

常见问题

译者注：这部分内容不翻译了，请参考原英文文档。

- Unable to Connect to Server
- Unable to Load the Browser
- Selenium Cannot Find the AUT
- Firefox Refused Shutdown While Preparing a Profile
- Versioning Problems
- Error message: “(Unsupported major.minor version 49.0)” while starting server
- 404 error when running the getNewBrowserSession command
- Permission Denied Error
- Handling Browser Popup Windows
- On Linux, why isn’t my Firefox browser session closing?

- Firefox *chrome doesn’t work with custom profile
- Is it ok to load a custom pop-up as the parent page is loading (i.e., before the parent page’s javascript window.onload() function runs)?
- Problems With Verify Commands
- Safari and MultiWindow Mode
- Firefox on Linux
- IE and Style Attributes
- Error encountered - “Cannot convert object to primitive value” with shut down of *googlechrome browser
- Where can I Ask Questions that Aren’t Answered Here?

Selenium Grid

快速上手

如果你对 Selenium 自动化测试已经非常熟悉，你仅仅需要一个快速上手来使程序运行起来。本章节的内容能满足不同的技术层次，但是如果你仅仅需要一个可以快速上手的指引，那么就显得有点多。如果是这样，你可以参考 Selenium Wiki 的相关文章。

什么是 Selenium-Grid ?

Selenium-Grid 允许你在多台机器的多个浏览器上并行的进行测试，也就是说，你可以同时运行多个测试。本质上来说就是，Selenium-Grid 支持分布式的测试执行。它可以让你的测试在一个分布式的执行环境中运行。

何时需要使用

通常，以下两种情况你都会需要使用 Selenium-Grid。

- 在多个浏览器中运行测试，在多个版本的浏览器中进行测试，或在不同操作系统的浏览器中进行测试。
- 减少测试运行时间。

Selenium-Grid 通过使用多台机器并行地运行测试来加速测试的执行过程。例如,如果你有一个包含100个测试用例的测试套件,你使用 Selenium-Grid 支持4台不同的机器（虚拟机或实体机均可）来运行那些测试，同仅使用一台机器相比，你的测试所需要的运行时间大致为其1/4。对于大型的测试套件和那些会进行大量数据校验的需要长时间运行的测试套件来说，这将节约很多时间。有些测试套件可能要运行好几小时。另一个需要缩短套件运行时间的原因是开发者检入（check-in）AUT 代码后，需要缩短测试的运行周期。越来越多的团队使用敏捷开发，相比整夜整夜的等待测试通过，他们希望尽快地看到测试反馈。

Selenium-Grid 也可以用于支持多执行环境的测试运行，典型的，同时在多个不同的浏览器中运行。例如，Grid 的虚拟机可以安装测试必须的各种浏览器。于是，机器 1 上有 ie8，机器 2 上有 ie9，机器 3 上有最新版的 chrome，而机器 4 上有最新版的 firefox。当测试套件运行时，Selenium-Grid 可以使测试在指定的浏览器中运行，并且接收每个浏览器的运行结果。

另外，我们可以拥有一个装有多个类型和版本都一样的浏览器 Grid。例如，一个 Grid 拥有 4 台机器，每台机器可以运行 3 个 firefox 12 实例，形成一个 firefox 的服务农场。当测试套件运行时，每个传递给 Selenium-Grid 的测试都被指派给下一个可用的 firefox 实例。通过这种

方式，我们可以使得同时有 12 个测试在并行的运行以完成测试，显著地缩短了测试完成需要的时间。

Selenium-Grid 非常灵活。以上两个例子可以联合起来使用，这样可以就可以使得不同类型和版本的浏览器有多个可运行实例。使用这样的配置，既并行地执行测试，同时又可以测试多个浏览器类型和版本。

Selenium-Grid 2.0

Selenium-Grid 2.0 是在编写本文时(5/26/2012)已发布的最新版本。它同版本 1 有很多不同之处。在 2.0 中，Selenium-Grid 和 Selenium-RC 服务端进行了合并。现在，你仅需要下载一个 jar 包就可以获得它们。

Selenium-Grid 1.0

版本 1 是 Selenium-Grid 的第一个发布版本。如果你是一个 Selenium-Grid 新手，你应该选择版本 2 。新版本已经在原有基础上进行了更新，页增加了一些新特性，并且支持 Selenium-WebDriver。一些老的系统可能仍然在使用版本 1.关于 Selenium-Grid 版本 1 的信息可以参考 Selenium-Grid website

Selenium-Grid 的 Hub 和 Nodes 是如何工作的？

Grid 由一个中心和一到多个节点组成。两者都是通过 selenium-server.jar 启动。在接下来的章节中，我们列出了一些例子。

中心接收要执行的测试信息，包括在哪些平台和浏览器执行等。它知道每个注册了的节点的配置。根据测试信息，它会选择符合需求的节点进行测试。一旦选定了一个节点，测试脚本就会初始化 Selenium 命令，并且由重心发送给选定的要运行测试的节点。这个节点会启动浏览器，然后在浏览器中执行这个 AUT 的 Selenium 命令。

我们提供了一些图标来演示其原理。第二张图标是用以说明 Selenium-Grid 1 的，版本 2 也适用并且对于我们的描述是一个很好的说明。唯一的区别在于相关术语。使用“Selenium-Grid 节点”替换“Selenium Remote Control”即符合我们对 Selenium-Grid 2 的描述。

下载

下载过程很简单。从 SeleniumHq 站点的下载页面下载 Selenium-Server jar 包。你需要的链接在“Selenium-Server (以前是 Selenium-RC)”章节中。

将它存放到任意文件夹中。你需要确保机器上正确的安装了 java。如果 java 没有正常运行，检查你系统的 path 变量是否包含了 java.exe 的路径。

启动 Selenium-Grid

由于节点对中心有依赖，所以你通常需要先启动一个中心。这也不是必须的，因为节点可以识别其中心是否已经启动，反之亦然。作为教程，我们建议你先启动中心，否则会显示一些错误信息，你应该不会想在第一次使用 Selenium-Grid 的时候就看到它们。

启动中心

通过在命令行执行以下命令，可以启动一个使用默认设置的中心。所有平台可用，包括 Windows Linux, 或 MacOs 。

```
java -jar selenium-server-standalone-2.21.0.jar -role hub
```

我们将在接下来的章节中解释各个参数。注意，你可能需要修改上述命令中 jar 包的版本号，这取决于你使用的 selenium-server 的版本。

启动节点

通过在命令行执行以下命令，可以你懂一个使用默认设置的节点。

```
java -jar selenium-server-standalone-2.21.0.jar -role node  -hub http://localhost:4444/gr
```

该操作假设中心是使用默认设置启动的。中心用于监听请求使用的默认端口号为 4444，这就是为什么端口 4444 被用于中心 url 中。同时“localhost”假定你的节点和中心运行在同一台机器上。对于新手来说，这是最简单的方式。如果要在两台不同的机器上运行中心和节点，只需要将“localhost”替换成中心所在机器的 hostname 即可。

警告： 确保运行中心和节点的机器均已关闭防火墙，否则你将看到一个连接错误。

配置 Selenium-Grid

默认配置

JSON 配置文件

通过命令行选项配置

中心配置

通过指定 `-role hub` 即以默认设置启动中心：

```
java -jar selenium-server-standalone-2.21.0.jar -role hub
```

你将看到以下日志输出：

```
Jul 19, 2012 10:46:21 AM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid server
2012-07-19 10:46:25.082:INFO:osjs.Server:jetty-7.x.y-SNAPSHOT
2012-07-19 10:46:25.151:INFO:osjsh.ContextHandler:started o.s.j.s.ServletContextHandler{/
2012-07-19 10:46:25.185:INFO:osjs.AbstractConnector:Started SocketConnector@0.0.0.0:4444
```

指定端口

中心默认使用的端口是 4444 。这是一个 TCP/IP 端口，被用于监听客户端，即自动化测试脚本到 Selenium-Grid 中心的连接。如果你电脑上的另一个应用已经占用这个接口，或者你已经启动了一个 Selenium-Server，你将看到以下输出：

```
10:56:35.490 WARN - Failed to start: SocketListener0@0.0.0.0:4444
Exception in thread "main" java.net.BindException: Selenium is already running on port 44
```

如果看到这个信息，你可以关掉在使用端口 4444 的进程，或者告诉 Selenium-Grid 使用一个别的端口来启动中心。 `-port` 选项用于修改中心的端口：

```
java -jar selenium-server-standalone-2.21.0.jar -role hub -port 4441
```

即使已经有一个中心运行在这台机器上，只要它们不使用同一个端口，就能正常工作。

你可能想知道哪个进程使用了 4444 端口，这样你就可以让中心使用这个默认端口。使用以下命令可以查看你机器上所有运行程序使用的端口：

```
netstat -a
```

Unix/Linux, MacOs 和 Windows 均支持此命令，只是在 Windows 中 -a 参数为必须的。基本上，你需要显示进程 id 和端口。在 Unix 中，你可以通过管道 “grep” 输出那些你关心的端口相关的条目。

节点配置

时间参数

获取命令行帮助

Selenium-Server 提供了一个可选项列表，每个选项都有一个简短的描述。目前（2012 夏），命令行帮助还有一些奇怪，但是如果你知道如何去找、如何解读信息会对你很有帮助。

Selenium-Server 提供了两种不同的功能，Selenium-RC server 和 Selenium-Grid。它们是两个不同的团队编写的，所以每个功能的命令行帮助被放置在不同的地方。因此，对于新手来说，在初次使用任意一个功能时，帮助都不是那么显而易见。

如果你仅传递一个 `-h` 选项，你将看到 Selenium-RC Server 的可选项而不是 Selenium-Grid 的。

```
java -jar selenium-server-standalone-2.21.0.jar -h
```

上述代码将显示 Selenium-RC server 选项。如果你想看到 Selenium-Grid 的命令行帮助，你需要先使用 `-hub` 或 `-node` 选项告诉 Selenium-Server 你想看的是关于 Selenium-Grid 的，然后再追加 `-h` 选项。

```
java -jar selenium-server-standalone-2.21.0.jar -role node -h
```

对于这个问题，你还可以给 `-role node` 传递一个垃圾参数：

```
java -jar selenium-server-standalone-2.21.0.jar -role node xx
```

你将先看到 “INFO...” 和一个 “ERROR”，在其后你将看到 Selenium-Grid 的命令行选项。我们没有列出这个命令的所有输出，因为它实在太长了，这个输出的最初几行看起来如下：

```
Jul 19, 2012 10:10:39 AM org.openqa.grid.selenium.GridLauncher main
INFO: Launching a selenium grid node
org.openqa.grid.common.exception.GridConfigurationException: You need to specify a hub to
        at org.openqa.grid.common.RegistrationRequest.validate(RegistrationRequest.java:6
        at org.openqa.grid.internal.utils.SelfRegisteringRemote.startRemoteServer(SelfReg
        at org.openqa.grid.selenium.GridLauncher.main(GridLauncher.java:72)
Error building the config :You need to specify a hub to register to using -hubHost X -hub
Usage :
  -hubConfig:
        (hub) a JSON file following grid2 format.

 -nodeTimeout:
        (node) <XXXX>  the timeout in seconds before the hub
          automatically ends a test that hasn't had aby activity than XX
          sec.The browser will be released for another test to use.This
          typically takes care of the client crashes.
```

常见错误

Unable to acess the jarfile

Unable to access jarfile selenium-server-standalone-2.21.0.jar

无论是启动中心还是节点都有可能产生这个错误。这意味着 java 无法找到 selenium-server jar 包。你需要从 selenium-server-XXXX.jar 文件存放在目录运行命令或者指定 jar 包的完整路径。

WebDriverJS

WebDriver 的 JavaScript 语言绑定。本文包含以下内容：

- 介绍
- 快速上手
  - 在 Node 中运行
  - 在浏览器中运行
- 设计细节
  - 管理异步 API
  - 同服务端通讯
  - /xdrpc
- 未来计划

介绍

WebDriver 的 JavaScript 绑定（WebDriverJS），可以使 JavaScript 开发人员避免上下文切换的开销，并且可以让他们使用和项目开发代码一样的语言来编写测试。WebDriverJS 既可以在服务端运行，例如 Node，也可以在浏览器中运行。

警告： WebDriverJS 要求开发者习惯异步编程。对于那些 JavaScript 新手来说可能会发现 WebDriverJS 有点难上手。

快速上手

在 Node 中运行

虽然 WebDriverJS 可以在 Node 中运行，但它至今还没有实现本地驱动的支持（也就是说，你的测试必须使用一个远程的 WebDriver 服务）。并且，你必须编译 Selenium 服务端，将其添加到 WebDriverJS 模块。进入 Selenium 客户端的根目录，执行：

```
$ ./go selenium-server-standalone //javascript/node:webdriver
```

当两个目标都被编译好以后，启动服务和 Node，开始编写测试代码：

```
$ java -jar build/java/server/src/org/openqa/grid/selenium/selenium-standalone.jar &
$ node

var webdriver = require('./build/javascript/node/webdriver');

var driver = new webdriver.Builder().
    usingServer('http://localhost:4444/wd/hub').
    withCapabilities({
      'browserName': 'chrome',
      'version': '',
      'platform': 'ANY',
      'javascriptEnabled': true
    }).
    build();

driver.get('http://www.google.com');
driver.findElement(webdriver.By.name('q')).sendKeys('webdriver');
driver.findElement(webdriver.By.name('btnG')).click();
driver.getTitle().then(function(title) {
  require('assert').equal('webdriver - Google Search', title);

});

driver.quit();
```

在浏览器中运行

除了 Node，WebDriverJS 也可以直接在浏览器中运行。编译比Node方式少很多依赖的浏览器模块，运行：

```
$ ./go //javascript/webdriver:webdriver
```

为了和可能不在同一个域下的 WebDriver 的服务端进行通信，客户端使用的是修改过的 JsonWireProtocol 和 cross-origin resource sharing ：

```
<!DOCTYPE html>
<script src="webdriver.js"></script>
<script>
  var client = new webdriver.http.CorsClient('http://localhost:4444/wd/hub');
  var executor = new webdriver.http.Executor(client);

  // 启动一个新浏览器，这个浏览器可以被这段脚本控制
  var driver = webdriver.WebDriver.createSession(executor, {
    'browserName': 'chrome',
    'version': '',
    'platform': 'ANY',
    'javascriptEnabled': true
  });

  driver.get('http://www.google.com');
  driver.findElement(webdriver.By.name('q')).sendKeys('webdriver');
  driver.findElement(webdriver.By.name('btnG')).click();
  driver.getTitle().then(function(title) {
    if (title !== 'webdriver - Google Search') {
      throw new Error(
          'Expected "webdriver - Google Search", but was "' + title + '"');
    }
  });

  driver.quit();
</script>
```

控制宿主浏览器

启动一个浏览器运行 WebDriver 来测试另一个浏览器看起来比较冗余（相比在 Node 中运行而言）。但是，使用 WebDriverJS 在浏览器中运行自动化测试是浏览器真实在跑这些脚本的。这只要服务端的 url 和浏览器的 session id 是已知的就可以实现。这些值可能会直接传递给 builder，它们也可以通过从页面 url 的查询字符串中解析出来的 wdurl 和 wdsid 定义 。

```
<!-- Assuming HTML URL is /test.html?wdurl=http://localhost:4444/wd/hub&wdsid=foo1234 -->
<!DOCTYPE html>
<script src="webdriver.js"></script>
<input id="input" type="text"/>
<script>
  // Attaches to the server and session controlling this browser.
  var driver = new webdriver.Builder().build();

  var input = driver.findElement(webdriver.By.tagName('input'));
  input.sendKeys('foo bar baz').then(function() {
    assertEquals('foo bar baz',
        document.getElementById('input').value);
  });
</script>
```

警告

在浏览器中使用 WebDriverJS 有几个需要注意的地方。首先，webdriver.Builder 类只能用于已存在的 session。为了获得一个新的 session，你必须像上面的例子那样手工创建。其次，有一些命令可能会影响运行 WebDriverJS 脚本的页面。

- webdriver.WebDriver#quit: quit 命令将终止整个浏览器进程，包括在运行 WebDriverJS 的窗口。除非你确定要这样做，否则不要使用这个命令。
- webdriver.WebDriver#get: WebDriver 的接口被设计为尽量接近用户的操作。这意味着无论 WebDriver 客户端当前聚焦在哪个帧，导航命令（如：driver.get(url)）总是指向最高层的帧。在操作宿主浏览器时，WebDriverJS 脚本可以通过使用 .get 命令导航离开当前页面，而当前页面仍然获得焦点。 如果要自动操作一个宿主浏览器但仍想在页面间跳转，请把WebDriver客户端的焦点设在另一个窗口上(这和Selenium RC 的多窗口模式的概念非常相似):

```
<!DOCTYPE html>
<script src="webdriver.js"></script>
<script>
  var testWindow = window.open('', 'slave');

  var driver = new webdriver.Builder().build();
  driver.switchTo().window('slave');
  driver.get('http://www.google.com');
  driver.findElement(webdriver.By.name('q')).sendKeys('webdriver');
  driver.findElement(webdriver.By.name('btnG')).click();
</script>
```

调试 **Tests**

你可以使用 WebDriver 的服务来调试在浏览器中使用 WebDriverJS 运行的测试。

```
$ ./go selenium-server-standalone
$ java -jar \
    -Dwebdriver.server.session.timeout=0 \
    build/java/server/src/org/openqa/grid/selenium/selenium-standalone.jar &
```

启动服务后，访问 WebDriver 的控制面板：http://localhost:4444/wd/hub。你可以使用这个控制面板查看，创建或者删除 sessions。选择一个要调试的 session 后，点击 “load script” 按钮。在弹出的对话框中，输入你的 WebDriverJS 测试的地址：服务端将在你的浏览器中打开这个页面，这个页面的 url 含有额外的参数用于 WebDriverJS 客户端和服务端的通讯。

支持的浏览器

- IE 8+
- Firefox 4+
- Chrome 12+
- Opera 12.0a+
- Android 4.0+

设计细节

管理异步 API

不同于其他那些提供了阻塞式 API 的语言绑定，WebDriverJS 完全是异步的。为了追踪每个命令的执行状态， WebDriverJS 对 “promise” 进行了扩展。promise 是一个这样的对象，它包含了在未来某一点可用的一个值。JavaScript 有几个 promise 的实现，WebDriverJS 的 promise 是基于 CommonJS 的 Promise/A 提议，它定义了 promise 是任意对象上的 then 函数属性。

```
/**
 * Registers listeners for when this instance is resolved.
 *
 * @param {?function(*)} callback The function to call if this promise is
 *     successfully resolved. The function should expect a single argument: the
 *     promise's resolved value.
 * @param {?function(*)=} opt_errback The function to call if this promise is
 *     rejected. The function should expect a single argument: the failure
 *     reason. While this argument is typically an {@code Error}, any type is
 *     permissible.
 * @return {!Promise} A new promise which will be resolved
 *     with the result of the invoked callback.
 */
Promise.prototype.then = function(callback, opt_errback) {
};
```

通过使用 promises，你可以将一连串的异步操作连接起来，确保每个操作执行时，它之前的操作都已经完成：

```
var driver = new webdriver.Builder().build();
driver.get('http://www.google.com').then(function() {
  return driver.findElement(webdriver.By.name('q')).then(function(searchBox){
    return searchBox.sendKeys('webdriver').then(function() {
      return driver.findElement(webdriver.By.name('btnG')).then(function(submitButton) {
        return submitButton.click().then(function() {
          return driver.getTitle().then(function(title) {
            assertEquals('webdriver - Google Search', title);
          });
        });
      });
    });
  });
});
```

不幸的是，上述范例非常冗长，难以辨别测试的意图。为了提供一套不降低测试可读性的干净利落的异步操作 API, WebDriverJS 引入了一个 promise “管理器” 来调度和执行所有的命令。

简言之，promise 管理器处理用户自定义任务的调度和执行。管理器保存了一个任务调度的列表，当列表中的某个任务执行完毕后，依次执行下一个任务。如果一个任务返回了一个 promise，管理器将把它当做一个回调注册，在这个 promise 完成后恢复其运行。WebDriver 将自动使用管理器，所以用户不需要使用链式调用。因此，之前的 google 搜索的例子可以简化成：

```
var driver = new webdriver.Builder().build();
driver.get('http://www.google.com');

var searchBox = driver.findElement(webdriver.By.name('q'));
searchBox.sendKeys('webdriver');

var submitButton = driver.findElement(webdriver.By.name('btnG'));
submitButton.click();

driver.getTitle().then(function(title) {
  assertEquals('webdriver - Google Search', title);
});
```

On Frames and Callbacks

就内部而言，promise 管理器保存了一个调用栈。在管理器执行循环的每一圈，它将从最顶层帧的队列中取一个任务来执行。任何被包含在之前命令的回调中的命令将被排列在一个新帧中，以确保它们能在所有早先排列的任务之前运行。这样做的结果是，如果你的测试是 written-in line，所有的回调都使用函数字面量定义，命令将按照它们在屏幕上出现的垂直顺序来执行。例如，考虑以下 WebDriverJS 测试用例：

```
driver.get(MY_APP_URL);
driver.getTitle().then(function(title) {
  if (title === 'Login page') {
    driver.findElement(webdriver.By.id('user')).sendKeys('bugs');
    driver.findElement(webdriver.By.id('pw')).sendKeys('bunny');
    driver.findElement(webdriver.By.id('login')).click();
  }
});
driver.findElement(webdriver.By.id('userPreferences')).click();
```

这个测试用例可以使用 WebDriver 的 Java API 重写如下：

```
driver.get(MY_APP_URL);
if ("Login Page".equals(driver.getTitle())) {
  driver.findElement(By.id("user")).sendKeys("bugs");
  driver.findElement(By.id("pw")).sendKeys("bunny");
  driver.findElement(By.id("login")).click();
}
driver.findElement(By.id("userPreferences")).click();
```

错误处理

既然所有 WebDriverJS 的操作都是异步执行的，我们就不能使用 try-catch 语句。取而代之的是，你必须为所有命令的 promise 返回注册一个错误处理的函数。这个错误处理函数可以抛出一个错误，在这种情况下，它将被传递给链中的下一个错误处理，或者他将返回一个不同的值来抑制这个错误并切换回回调处理链。

如果错误处理器没有正确的处理被拒绝的 promise（不只是哪些来自于 WebDriver 命令的），则这个错误会传播至错误处理链的父级帧。如果一个错误没有被抑制而传播到了顶层帧，promise 管理器要么触发一个 uncaughtException 事件（如果有注册监听的话），或者将错误抛给全局错误处理器。在这两种情况下，promise 管理器都将抛弃所有队列中后续的命令。

```
// 注册一个事件监听未处理的错误
webdriver.promise.Application.
    getInstance().
    on('uncaughtException', function(e) {
      console.error('There was an uncaught exception: ' + e.message);
    });

driver.switchTo().window('foo').then(null, function(e) {
  // 忽略 NoSuchWindow 错误，让其他类型的错误继续向上冒泡
  if (e.code !== bot.ErrorCode.NO_SUCH_WINDOW) {
    throw e;
  }
});
// 如果上面的错误不被抑制的话，这句将永远不会执行
driver.getTitle();
```

同服务端通讯

当在服务端环境中运行时，客户端不受安全沙箱的约束，可以简单的发送 http 请求（例如：node 的 http.ClientRequest）。当在浏览器端运行时，WebDriverJS 客户端就会收到同源策略的约束。为了和可能不在同一个域下的服务端通讯，WebDriverJS 客户端使用的是修改过的 JsonWireProtocol 和 cross-origin resource sharing。

Cross-Origin Resource Sharing

如果一个浏览器支持 cross-origin resource sharing (CORS), WebDriverJS 将使用 cross-origin XMLHttpRequests (XDR) 发送命令给服务端。服务端要想支持 XDR，就需要响应 preflight 请求，并带有合适的 access-control 头。

```
Access-Control-Origin: *
Access-Control-Allow-Methods: DELETE,GET,HEAD,POST
Access-Control-Allow-Headers: Accept,Content-Type
```

在编写本文时，已有 Firefox 4+, Chrome 12+, Safari 4+, Mobile Safari 3.2+, Android 2.1+, Opera 12.0a, 和 IE8+ 支持 CORS。不幸的是，这些浏览器的实现并不一致，也不是完全都遵循 W3C 的规范。

- IE 的 XDomainRequest 对象，比其 XMLHttpRequest 对象的功能要弱。XDomainRequest 只能发送哪些标准的 form 表单可以发送的请求。这限制了 IE 只能发送 get 和 post 请求（wire 协议要求支持 delete 请求）。
- WebKit 的 CORS 实现禁止了跨域请求的重定向，即使 access-control 头被正确设置了也是如此。
- 如果返回一个服务端错误（4xx 或 5xx），IE 和 Opera 的实现将触发 XDomainRequest/XMLHttpRequest 对象的错误处理，但是拿不到服务端返回的信息。这使得它们无法处理以标准的 JSON 格式返回的错误信息。

为了弥补这些短处，当在浏览器中运行时，WebDriverJS 将使用修改过的 JsonWireProtocol 和通过 /xdrpc 路由所有的命令。

/xdrpc

POST /xdrpc

作为命令的代理，所有命令相关的内容必须被编码成 JSON 格式。命令的执行结果将在 HTTP 200 响应中作为一个标准的响应结果返回。客户端依赖于响应的转台吗以确认命令是否执行成功。

参数：

- method - {string} http 方法
- path - {string} 命令路径
- data - {Object} JSON 格式的命令参数

返回：

{*} 命令执行的结果。

举个例子，考虑以下 /xdrpc 命令：

```
POST /xdrpc HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 94

{"method":"POST","path":"/session/123/element/0a/element","data":{"using":"id","value":"f
```

服务端将编码这个命令并重新分发：

```
POST /session/123/element/0a/element HTTP/1.1
Accept: application/json
Content-Type: application/json
Content-Length: 28

{"using":"id","value":"foo"}
```

不管是否成功，命令的执行结果都将作为一个标准的 JSON 返回：

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 60

{"status":7,"value":{"message":"Unable to locate element."}}
```

未来计划

以下是一些预期要做的事情。但什么时候完成，在现在仍然未知。如果你有兴趣参与开发，请加入 selenium-developers@googlegroups.com。当然，这是一个开源软件，你完全不需要等待我们。如果你有好主意，就马上开工吧：）

- 使用 AutomationAtoms 实现一个纯 JavaScript 的命令执行器。这将允许开发者使用 js 编写非常轻量的测试代码，并且可以运行在任何服务浏览器中（当然，仍然会收到同源策略的限制）。
- 基于扩展实现一个 SafariDriver。
- 为 Node 提供本地浏览器支持，而不需要通过 WebDriver Server 运行。

高级用户交互

原文：https://code.google.com/p/selenium/wiki/AdvancedUserInteractions

入门

高级用户交互API提供了一个更新更完善的机制来定义并描述用户在一个网页上的各种操作。这些操作包括：拖拽、按住CTRL键选择多个元素等等。

快速上手

为了生成一连串的动作，我们使用Actions来建立。首先，我们先配置操作：

```
Actions builder = new Actions(driver);

builder.keyDown(Keys.CONTROL)
.click(someElement)
.click(someOtherElement)
.keyUp(Keys.CONTROL);
```

然后，获得操作（Action）：

```
Action selectMultiple = builder.build();
```

最后，执行这个动作：

```
selectMultiple.perform();
```

这一系列的动作应该尽量的短。在使用中最好在执行一个简短的动作后验证页面是否处于正确的状态，然后再执行下面的动作。下一节将会列出所有可用的动作（Action），并且说明它们如何进行扩展。

键盘交互（**Keyboard interactions**）

键盘交互是发生在一个特定的页面元素的，而webdriver会确保这个页面元素在执行键盘动作时处于正确的状态。这个正确的状态，包括页面元素滚动到可视区域并定位到这个页面元素。
既然这个新的API是面向用户（user-oriental）的接口，那么对于一个用户，在对一个元素输入文本前做显式的交互就更加的符合逻辑。这意味着，当想定位到相邻的页面元素时，可能需要点击一下元素或按下Tab（`Keys.TAB`）键。

The new interactions API will (first) support keyboard actions without a provided element. The additional work to focus on an element before sending it keyboard events will be added later on.

鼠标交互（**Mouse interactions**）

鼠标操作有一个上下文-鼠标的当前位置。因此，当为几个鼠标操作设定一个上下文时，第一个操作的上下文就是元素的相对位置，下一个操作的上下文就上一个操作后的鼠标相对位置。

支持情况

这个针对操作以及动作生成器的API已经（绝大部分）完成。HtmlUnit和Firefox已经完全支持，Opera和IE正在支持中。

大纲

单个动作

所有的动作都实现了 `Action` 接口，这个接口只有一个方法： `perform()` 。每个动作所需要的信息都通过Constructor传入。当调用这个动作的时候，动作知道如何与页面交互（如，找到活动的元素并输入文本或者计算出在屏幕上的点击坐标）并且调用底层实现来实现这个交互。
下面是一些动作：

- ButtonReleaseAction - 释放鼠标左键
- ClickAction - 相当于 WebElement.click()
- ClickAndHoldAction - 按下鼠标左键并保持
- ContextClickAction - 一般就是鼠标右键，通常调出右键菜单用。
- DoubleClickAction - 双击某个元素
- KeyDownAction - 按下修饰键（SHIFT，CTRL，ALT，WIN）
- KeyUpAction - 释放修饰键
- MoveMouseAction - 移动鼠标从当前位置到另外的元素.
- MoveToOffsetAction - 移动鼠标到一个元素的偏移位置（偏移可以为负，元素是鼠标刚移动到的那个元素）。
- SendKeysAction - 相当于 WebElement.sendKey(...)

`CompositeAction` 包含一系列的动作，当被调用的时候，它会调用它所包含的所有动作的perform方法。通常，这个动作通常都不是直接建立的，一般是使用 `ActionChainsGenerator` 。

生成动作链

`Actions` 链生成器实现了创建者模式来新建一个包含一组动作的 `CompositeAction` 。使用Actions生成器可以很容易的生成动作并调用 `build ()` 方法来获得复杂的操作。

```
Actions builder = new Actions(driver);

Action dragAndDrop = builder.clickAndHold(someElement)
   .moveToElement(otherElement)
   .release(otherElement)
   .build();

dragAndDrop.perform();
```

有一个对 `Actions` 进行扩展的计划，给 `Actions` 类添加一个方法，这个方法可以追加任何动作到其拥有的动作列表上。这将允许添加扩展的动作，而不用人工创建CompositeAction。关于扩展 `Actions` ,请往下看。

扩展Action接口的指导

Action接口只有一个方法- `perform()` 。除了实际的交互本身，所有的条件判断也都应该在这个这个方法里实现。在动作创建和动作实际执行这段时间内，很可能页面的状态已经发生了变化，比如元素的可视情况已经坐标已经不能找到了。

实现细节

为了达到每个操作的执行与具体实现的分离，所有的动作都依赖2个接口： `Mouse` 和 `Keyboard` 。这些接口被所有支持高级用户接口API的driver实现了。需要注意的是，这些接口是为了让动作使用的，而不是最终用户。本节的信息，主要是针对想扩展WebDriver的开发者的。

一字警告

`Keyboard` 和 `Mouse` 接口是设计用来支持各种Action类的。有鉴于此，它们的API没有Actions链生成器API稳定。直接使用这些接口可能达不到期望的结果，因为Action类做了额外的工作来确保在实际事件触发时处于正确的环境条件。这些准备工作包括定位在正确的元素上或者鼠标交互前保证元素是可见的。

Keyboard接口

Keyboard接口包含3个方法：

- void sendKeys(CharSequence... keysToSend) - 与 sendKeys(...)相似.
- void pressKey(Keys keyToPress) - 按下一个键并保持。键仅限于修饰键(Control, Alt and

Shift).
- void releaseKey(Keys keyToRelease) - 释放修饰键.

至于如何在调用之间保存修饰键的状态是Keyboard接口实现类的职责。只有活跃的元素才会接收到这些事件。

Mouse接口

`Mouse` 接口包含以下方法（有可能不久之后会有变化）：

- void click(WebElement onElement) - 同click()方法一样.
- void doubleClick(WebElement onElement) - 双击一个元素.
- void mouseDown(WebElement onElement) - 在一个元素上按下左键并保持 Action selectMultiple = builder.build();
- void mouseUp(WebElement onElement) - 在一个元素上释放左键.
- void mouseMove(WebElement toElement) - 从当前位置移动到一个元素
- void mouseMove(WebElement toElement, long xOffset, long yOffset) - 从当前位置移动到一个元素的偏移坐标
- void contextClick(WebElement onElement) - 在一个元素上做一个右键操作

Native events（原生事件） VS synthetic events（合成事件）

WebDriver提供的高级用户接口，要么是直接模拟的Javascript事件（即合成事件），要么就是让浏览器生成Javascript事件（即原生事件）。原生事件能更好的模拟用户交互，而合成事件则是平台独立的，这使得使用了替代的窗口管理器的linux系统显得尤其重要，具体参加native events on Linux。原生事件无论什么时候总是应该尽可能的使用。

下面的表格展示了浏览器对事件的支持情况。

浏览器	操作系统	原生事件	合成事件
Firefox	Linux	支持	支持（默认）
Firefox	Windows	支持（默认）	支持
Internet Explorer	Windows	支持（默认）	不支持
Chrome	Linux/Windows	支持*	不支持
Opera	Linux/Windows	支持（默认）	不支持
HtmlUnit	Linux/Windows	支持（默认）	不支持

ChromeDriver提供了2种模式来支持原生事件：Webkit事件和原始事件。其中Webkit事件是使用Webkit函数来触发的Javascript事件，而原始事件模式则使用的是操作系统级别的事件。

FirefoxDriver中，原生事件可以使用FirefoxProfile来进行开关控制。

```
FirefoxProfile profile = new FirefoxProfile();
profile.setEnableNativeEvents(true);
FirefoxDriver driver = new FirefoxDriver(profile);
```

例子

以下是原生事件与合成事件表现不同的一些例子：

- 使用合成事件，点击隐藏在其他元素下面的元素是可以的。使用原生事件，浏览器会将点击事件作用在所给坐标最外层的元素上，就像是用户点击在特定的位置一样。
- 当一个用户，按下TAB键希望焦点从当前元素定位到下一个元素，浏览器是可以做到的。使用合成事件的话，浏览器并不知道TAB键被按下了，因此也不会改变焦点。而使用原生事件，浏览器则会表现正确。